(2)

SRL-12-F-1993

COMPACT LIGHTWEIGHT CO_2 LASER FOR SDIO APPLICATION

Prepared by

Dr. Stephen Fulghum
Dr. Jonah Jacob

SCIENCE RESEARCH LABORATORY, INC.
15 Ward Street
Somerville, MA 02143
(617) 547-1122

November 16, 1993

PRELIMINARY FINAL TECHNICAL REPORT

Period for July 1, 1990 to June 30, 1992

Contract Number N00014-90-C-0162

Prepared for

OFFICE OF NAVAL RESEARCH
800 North Quincy Street
Arlington, VA 22217-5000

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE 11-15-93	3. REPORT TYPE AND DATES COVERED Final 7/1/90 to 6/30/92

4. TITLE AND SUBTITLE
Compact Lightweight CO_2 Laser for SDIO Applications

5. FUNDING NUMBERS
AA 9700400.2504 025
RA464 0 068342 2D 63220C
16021000S40K R&T #s
400223sre01, FRC:s40k
dtd 90 90JAN 31 (SDIO
Funds)

6. AUTHOR(S)
Dr. Stephen Fulghum
Dr. Jonah Jacob

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Science Research Laboratory, Inc
15 Ward St.
Somerville MA 02143

8. PERFORMING ORGANIZATION REPORT NUMBER
SRL-12-F-1993

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Naval Research
Department of the Navy
800 N. Quincy Street
Arlington, VA 22217

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for Public Release: Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

SRL has combined two all-solid-state pulsed power generators based on SCR-commutated nonlinear magnetic pulse compressors with low-pressure, conduction-cooled, CO_2 laser cavities in a program to produce high repetition rate laser systems without the weight penalty of flow loop cooling. The first CO_2 system used a transverse discharge, rectangular slab geometry, laser amplifier matched to the 10 Hz, SRL COLD-I solid-state pulser. Visible interferometric probes of the slab geometry laser medium showed the characteristic cylindrical wavefront error expected of a uniformly heated slab cooled at its walls. Conduction cooling of the 2.4 cm slab was sufficient to dissipate the thermal energy deposited by a single pulse dissipated to below the measurement level within 100 ms (10 Hz). The second CO_2 system used a longitudinal discharge, cylindrical geometry, laser amplifier combined with the new SSLAM-VIII pulser which produces 72 KV, 30 J pulses at up to 5 KHz. The longitudinal laser output power typically peaked at 50 Hz since the heat load from higher repetition rates could not be dissipated sufficiently rapidly from the 4 cm diameter discharge. Due to an impedance mismatch the maximum single pulse energy in steady state operation was 350 mJ (18 W average power). A circuit to provide proper matching has been designed but has not yet been implemented.

14. SUBJECT TERMS

15. NUMBER OF PAGES 50

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	None

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Overview

Science Research Laboratory has developed a family of all-solid-state pulsed power generators featuring levels of efficiency, reliability and power density far above what had been previously obtainable. A common feature of these systems is SCR-commutated nonlinear magnetic pulse compressors. Using SCRs rather than the more traditional thyratrons decreases system weight and volume while providing a significant increase in reliability. Pulsers using this technology have demonstrated lifetimes of 10^{10} shots.

The primary objective of this SBIR is to find ways of combining these pulsed power systems with new forms of CO_2 laser amplifiers as the first step in the development of a new family of CO_2 laser radar systems suitable for space-based applications. The CO_2 lasers investigated in this SBIR feature conduction-cooling of the laser mixture rather than the more typical flow loops and heat exchangers. Conduction cooling has significant size, weight and system complexity advantages which complement those of the all-solid-state pulsed power.

1.2 Program Plan

In this SBIR we built two forms of conduction-cooled CO_2 lasers matched to separate forms of SRL's solid-state pulsers. The first system uses a spark gap preionized, transverse discharge, rectangular slab geometry laser amplifier matched to an existing solid-state pulser (COLD-I). The laser medium in this slab system was interferometrically probed with a pulsed laser to test the effectiveness of its conduction cooling. The second system uses a longitudinal discharge, cylindrical geometry laser amplifier powered by a new pulsed power system (SSLAM-VIII) capable of much higher output power. This new pulsed power system features separate modules which individually produce 12 kV, 5 J pulses but which can be stacked to provide the 72 kV, 30 J pulses required for the longitudinal discharge. The output power of this second system was optimized by varying laser gas mixtures, pressure and pulse repetition rate.

1.3 Summary of Results

The interferometric measurements on the transverse discharge, slab geometry lasers showed that the thermal energy deposited by a single pulse dissipated completely within 100 ms. This means that, with actively cooled walls, the system could be run continuously at 10 Hz without the heat from one pulse affecting the next. At 50 ms the interferometric path error was only a small fraction of a visible wave. The test fringes showed the characteristic cylindrical error expected of a slab which is heated uniformly and cooled at its walls. This indicates that the system could have been run at 20 Hz with only a small effect on laser beam quality. The existing pulsed power supply repetition rate, however, was limited to 10 Hz.

The longitudinal discharge, cylindrical geometry laser was run successfully at repetition rates of up to 100 Hz. Its average output power typically peaked at 50 Hz, depending on the gas mixture and pressure. The 50 Hz practical limit was expected as calculations had indicated that the core of the discharge would be heated excessively above that rate. The efficiency of conversion from energy deposited in the discharge to laser output energy also varied with gas mixture and pressure but could be brought up to about 20%. The maximum pulse energy obtained, however, was only 350 mJ for a maximum average power of 18 W. The essential problem is that the optimum load for the SSLAM-VIII pulser was measured to be about 4 Ω whereas the typical discharge impedance ran in the 200 to 400 Ω range. A likely solution to this problem has been suggested but not yet implemented.

2. TRANSVERSE DISCHARGE SYSTEM TESTS

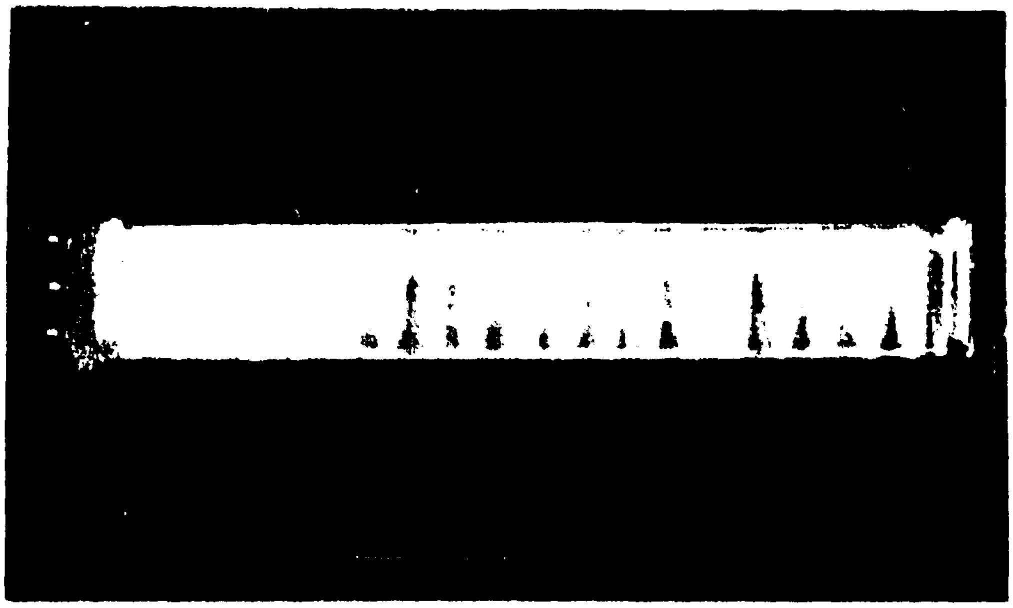
2.1 Laser Design Summary

While the new SSLAM-VIII pulsed power system was being built for this project, tests of conduction cooling were performed on a different CO_2 laser prototype. These experiments utilized a new version of a conduction-cooled, transverse discharge, slab geometry CO_2 laser system. An earlier version of this system had been produced under a previous contract(1) but proper tests of its medium quality had not been completed. This report will describe interferometric tests of the laser medium quality for the new version of the slab geometry, conduction-cooled laser.

Both lasers utilize a cylindrical housing to contain both the discharge electrodes and the low pressure (30 Torr) laser gas mixture required for conduction cooling. Two glass plate walls separate the discharge electrodes and serve to constrain the discharge within the enclosed, rectangular slab volume. In both systems the dimensions of the active volume were 13.5 cm in the direction of the discharge, 2.4 cm between the two glass plates and 100 cm along the optical cavity axis (a total of 3.2 liters). The distinctive slab geometry of this laser results from the requirement that heat produced in the discharge be carried by conduction to the laser walls. Figure 1 shows a photograph of the discharge produced by the new system along with a number of the discharge parameters. If the walls are too far apart the heat cannot be dissipated before the next pulse. If the walls are too close the total gas volume is insufficient to provide the required energy per pulse. A detailed discussion of the constraints for a conduction-cooled system are given in the proposal for this Phase II SBIR.(2)

The new version of the system employed several improvements over the original design. The first version of the laser had utilized a large number of individual cables feeding through

CONDUCTION COOLED CO_2 LASER DISCHARGE



Active Volume	$2.4 \times 13.5 \times 100$ cm^3
Voltage	5 kV
Electrode Spacing	13.5 cm
Current	800 Amps
Discharge Pulse Length	3 μs
Laser Pulse Length	40 μs
Overall Efficiency	14%
Total Gas Pressure	30 Torr

Figure 1

the cylinder wall to power the spark gap preionizers spaced along one of the electrodes. Those cables tended to leak which compromised the laser gas purity. The new version of the system utilized only two cable feedthroughs and each spark pin is individually ballasted.

2.2 Interferometric Probe Design

The physics of the conduction cooling process is sufficiently straightforward that its limitations were well understood. The interferometric tests on the laser medium were primarily designed to address the question of whether or not there would be high order distortions due to residual sound waves or shock waves in the medium left over from previous shots. An interferometric probe was built to test for these distortions.

The interferometric diagnostic consists of a Mach-Zehnder interferometer and a spatially filtered, pulsed dye laser probe beam. The dye laser pulse length is only 3 ns so the interferometric probe is a "snapshot" of the state of the laser medium at the time of the probe. A digital delay generator was used to sweep the time of the probe laser relative to the CO_2 laser discharge on successive shots. A CCD camera and digital framegrabber was used to record the fringes produced. The wavefront which bests fits these fringes was determined with an algorithm developed at SRL. This algorithm, presented in the Appendix, calculates a wavefront defined over an NxN grid. A minimum surface area constraint on the fit wavefront defines those grid values which are not directly determined from the fringes.

2.3 Interferometric Probe Data

Figure 2 shows a partial set of the fringe data taken. Figures 3 to 7 show the wavefront obtained from these fringes using an iterative, grid fit algorithm developed by SRL (listed in the Appendix).

The top graph in Fig. 3 shows the fringes taken just before a discharge pulse with a wall temperature roughly equal to the initial gas temperature. It takes a few microseconds for the gas

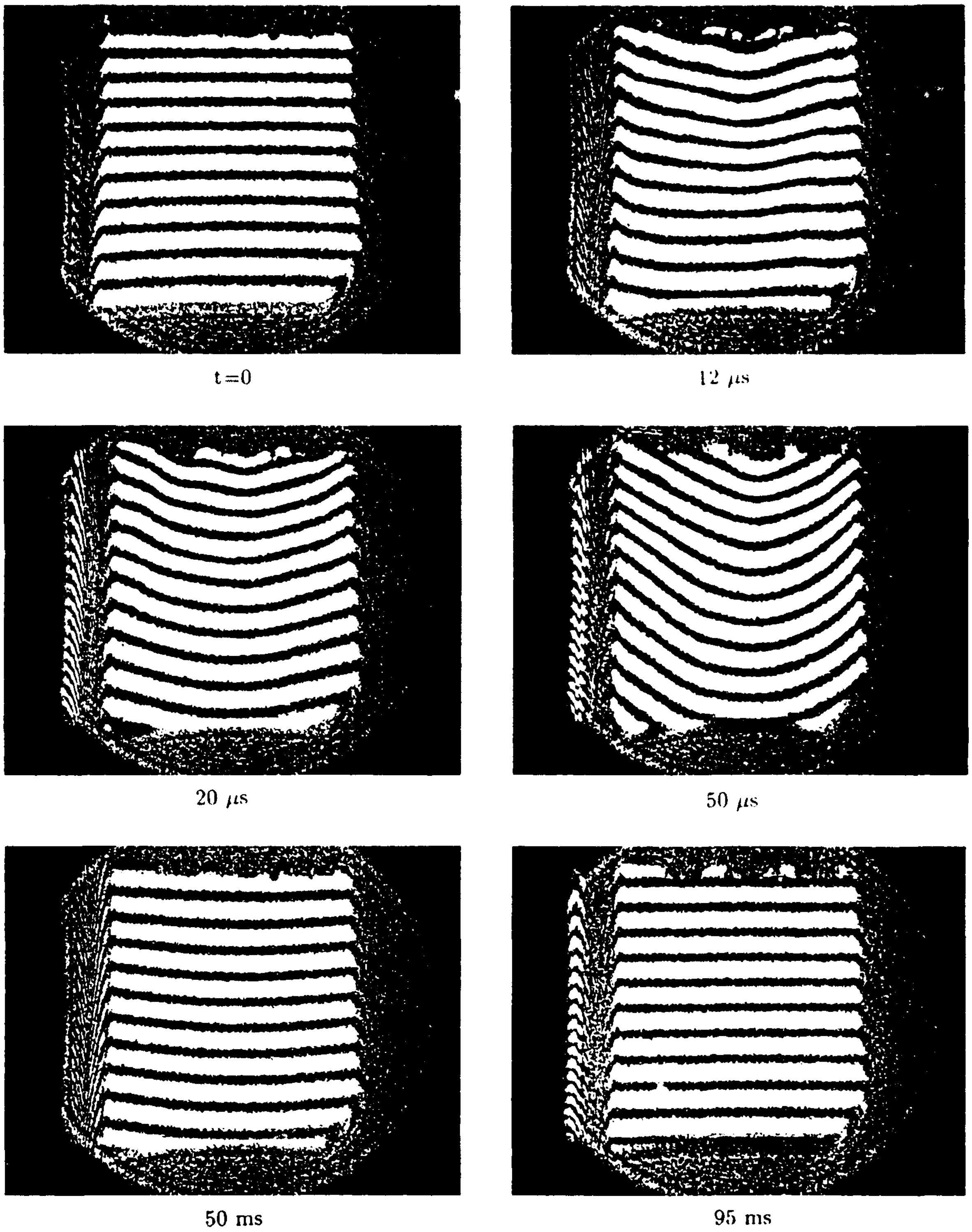


Figure 2: Interferometric probes of the transverse discharge, conduction cooled, CO_2 laser medium taken at the spark preionizer end of the discharge.

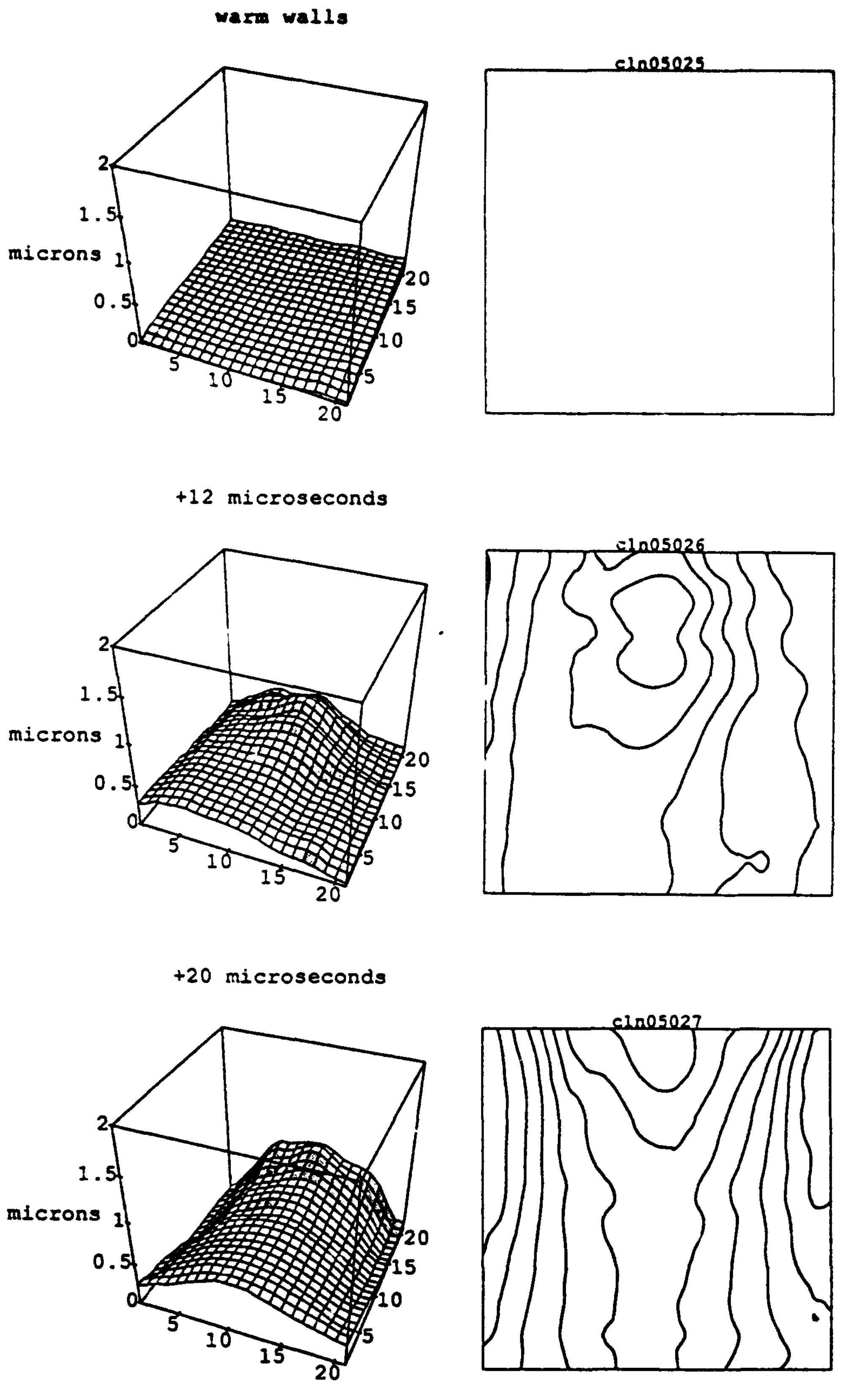


Figure 3: Optical Path Difference (OPD) for a single pass through the medium taken immediately following the discharge.

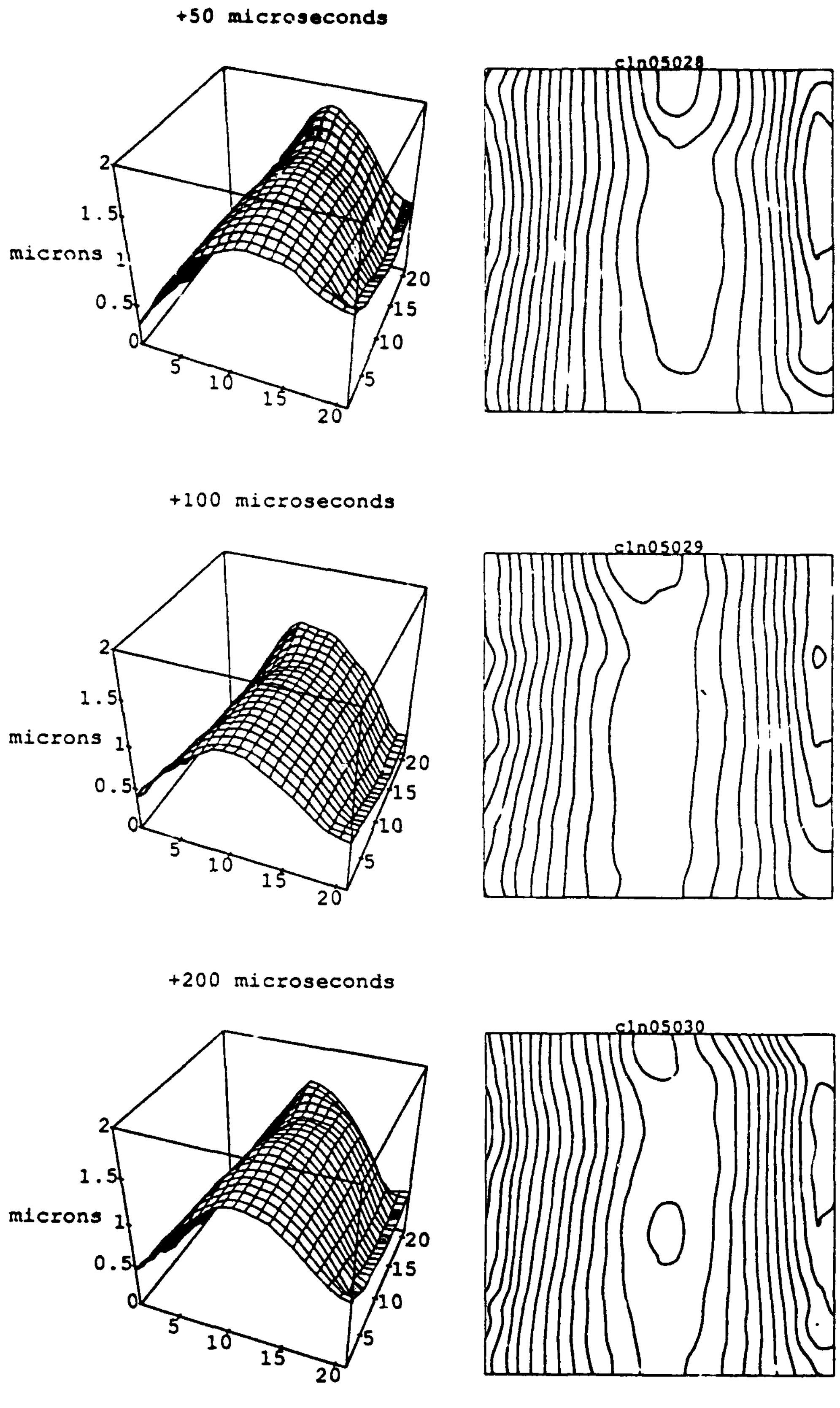


Figure 4: The OPD assumes its characteristic cylindrical shape within $50\mu s$.

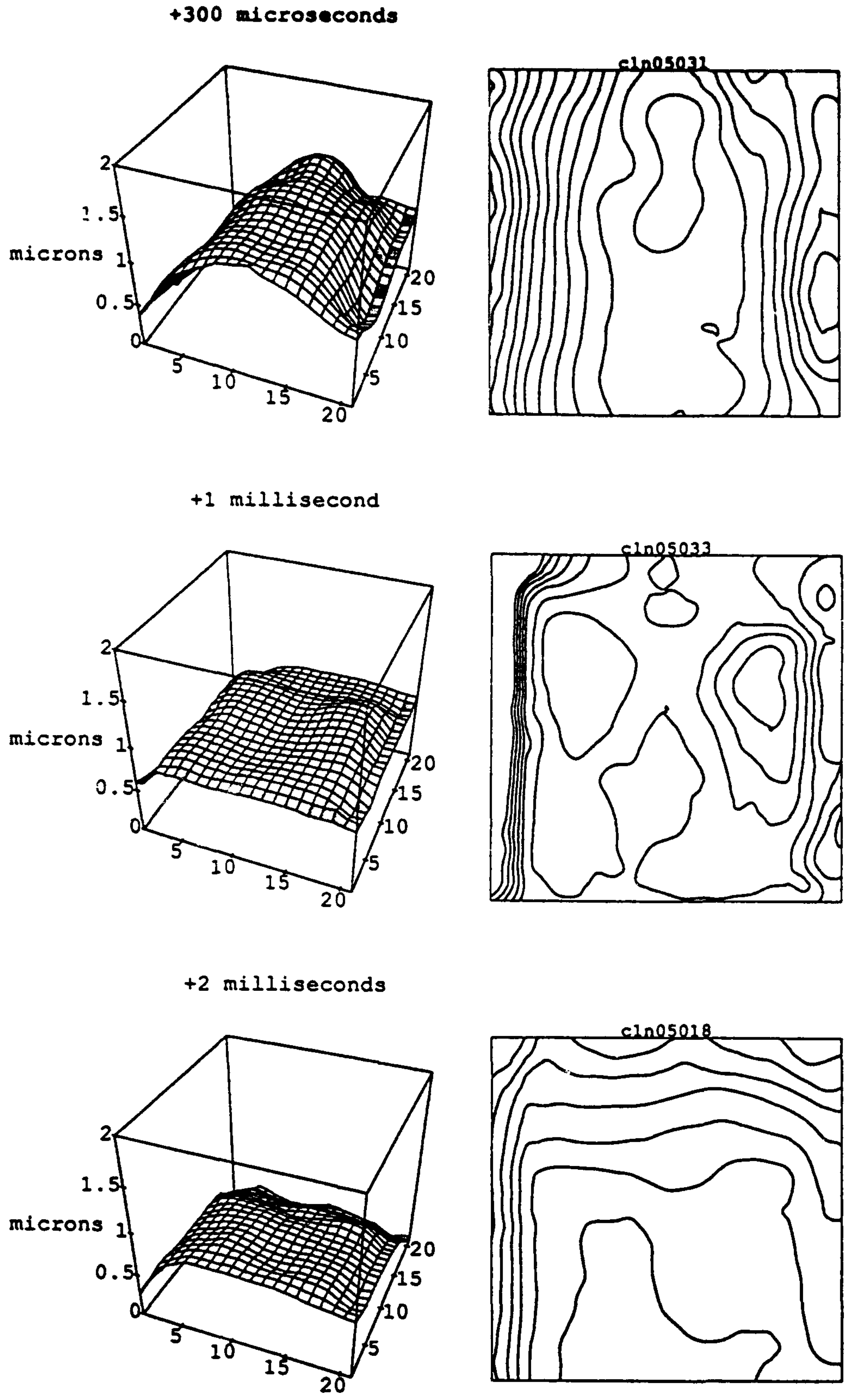


Figure 5: The OPD structure is complicated in the 1 ms time regime by stored energy flowing out of vibrational modes.

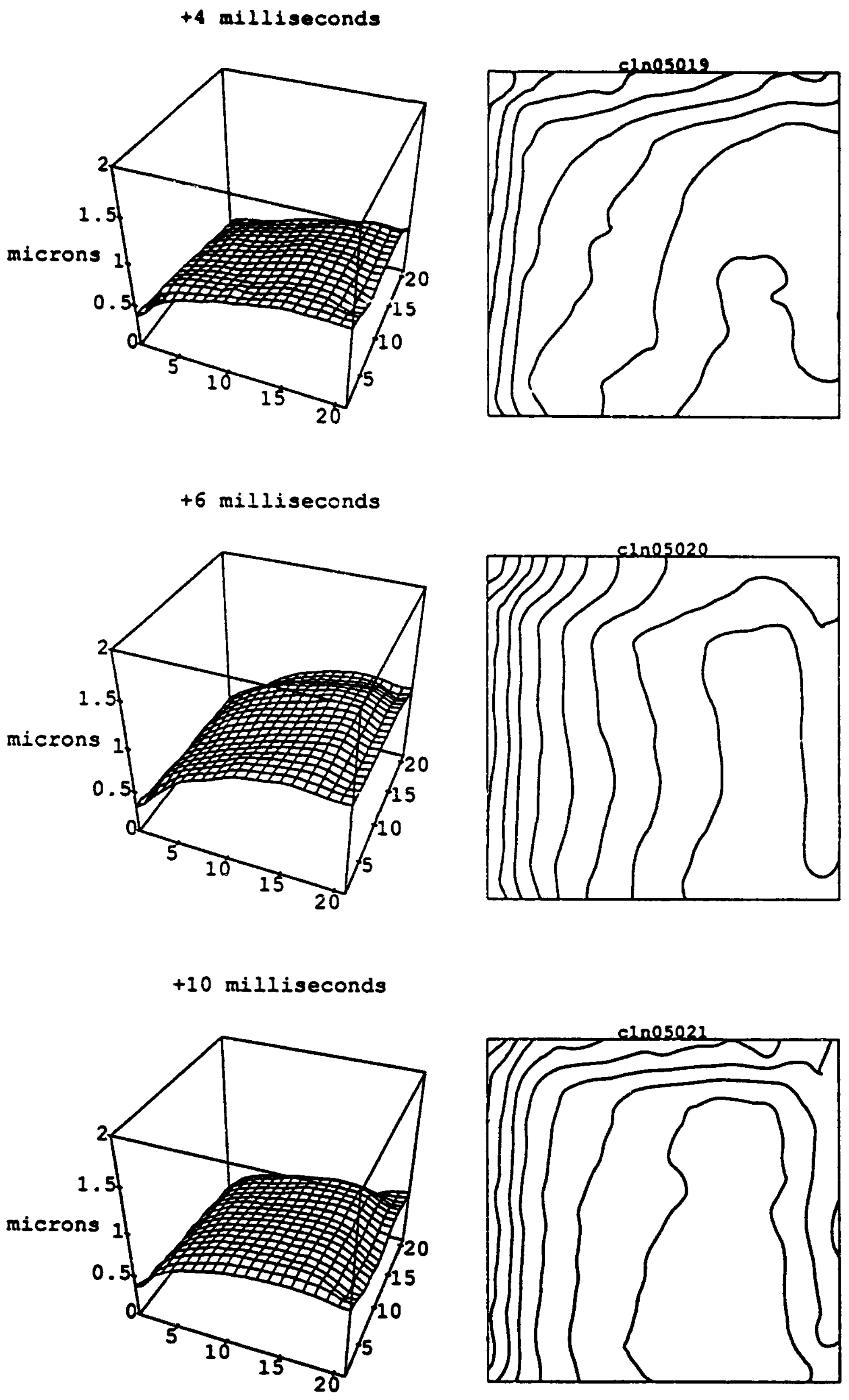


Figure 6: At a 10 ms delay (100 Hz) there is still a 1 μm single pass OPD which would add to the error of successive 100 Hz pulses.

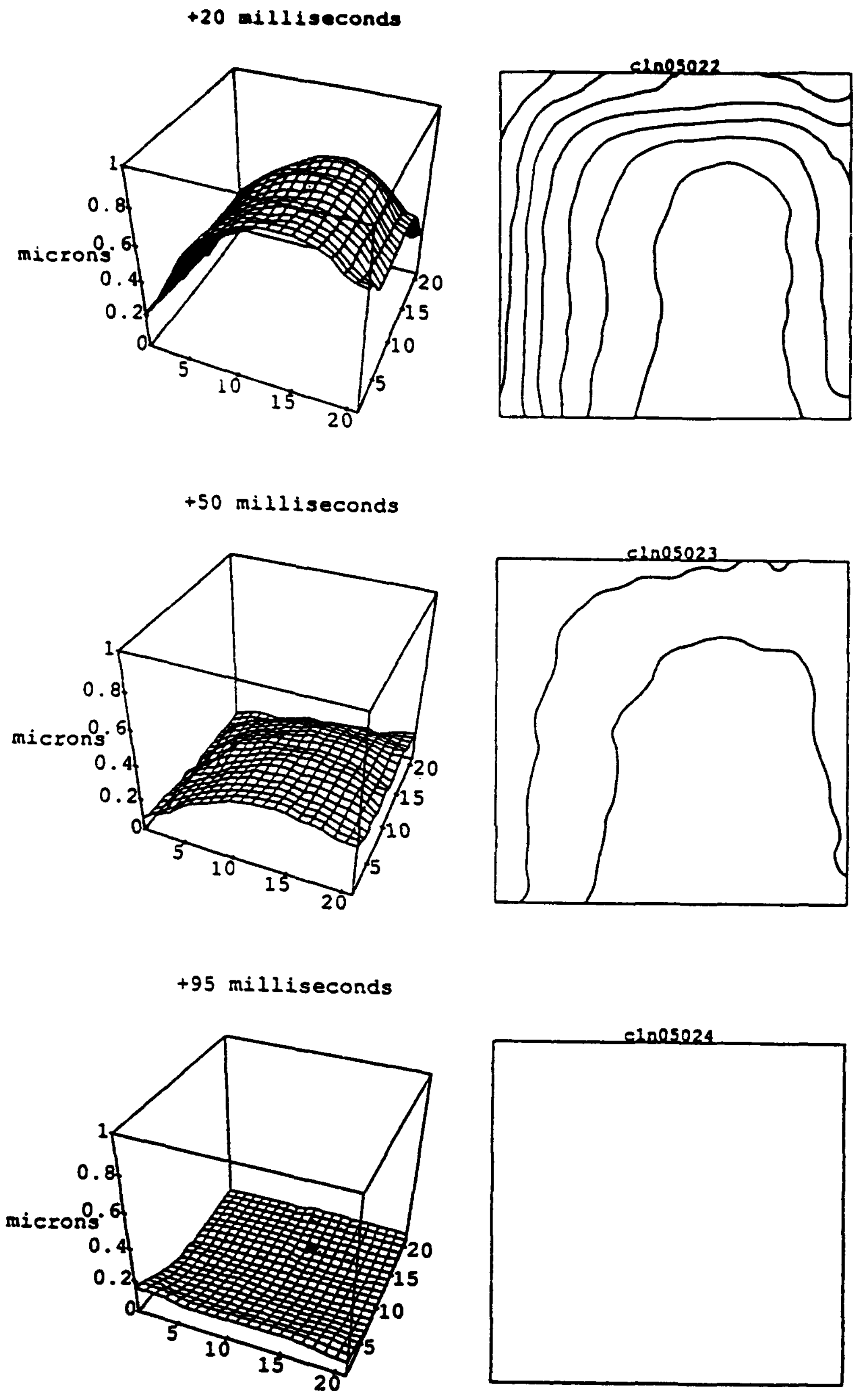


Figure 7: The OPD at 20 ms (50 Hz) is only about 1/20 th of a CO_2 laser wavelength. At a 10 Hz delay the error was not measurable with this system.

heated by the discharge to expand. The optical path difference (OPD) thus does not reach its maximum value for about 50 μs. In the 1 ms time range the OPD is complicated by what appears to be cooling in the center of the discharge channel. This effect is likely caused by energy stored in vibrational modes of the gases involved and, although interesting, does not affect the laser in any practical way. The OPD at 20 ms shown in Fig. 6 is the error which the laser pulse would see on the next pulse at a 50 Hz repetition rate. The error of successive pulse add so that the error on the third pulse at 50 Hz would be about the error at 20 ms plus the error at 50 ms. The total error at 50 Hz would thus still be below 1/10 of a wave at the CO_2 wavelength. The slight rise in temperature in the center of the cavity will cause the discharge to strike preferentially at that point. This feedback effect was noted in the longitudinal laser system to be described in the next section.

The transverse discharge slab laser could only be fired at a maximum rate of 10 Hz due to the choice of capacitors used in COLD-I. The OPD error at 95 ms shown in Fig. 7 was essentially at or below the measurement *limit* of the interferometer used. This means that the transverse discharge system always appeared to be running in the single pulse mode as far as the laser medium quality was concerned.

The experimental plan was to continue the measurements on the longitudinal discharge, conduction cooled laser system which could be run at 100 Hz and above. In practice, however, the efficiency of the energy deposition in the longitudinal discharge was too low to provide a good test of the conduction cooling in that geometry.

3. LONGITUDINAL DISCHARGE SYSTEM TESTS

3.1 SSLAM-VIII Pulsed Power System

The decision to develop a longitudinal discharge laser system was prompted by the significant advantages of the SSLAM series of pulsed power generators over more conventional systems. These pulsed power supplies are nearly indestructible, require an input voltage of only 600 V and can operate continuously at 5 kHz. In the SSLAM series a high voltage is induced on an isolated conducting rod using techniques borrowed from induction accelerators. The resulting power supply can be built in a modular fashion with successive modules simply adding another step in voltage. A longitudinal discharge requires these high voltages and is thus an obvious candidate for the application of this system.

Figure 8 shows a single SSLAM-VIII module which can produce 5 J pulses at 12 kV at repetition rates of up to 5 kHz. The module measures 7 inches in height by 17 inches in width and 28 inches in depth and weighs 39 kg. Figure 9 shows a close up of the two induction rods in each module and how they are connected in series to produce a total of 12 kV. Figure 10 shows the rack panel containing six 12 kV modules which produces a total pulse energy of 30 J at 72 kV.

3.2 Laser Design Summary

To match the high voltages which could be obtained from SSLAM-VIII, a longitudinal discharge laser was designed. The discharge cell consists of a 100 cm long thin wall glass tube with a 4.13 cm inside diameter. The voltages available from SSLAM-VIII are sufficient to break down the gas at pressures of up to about 30 Torr, which are also the pressures appropriate for conduction cooling.

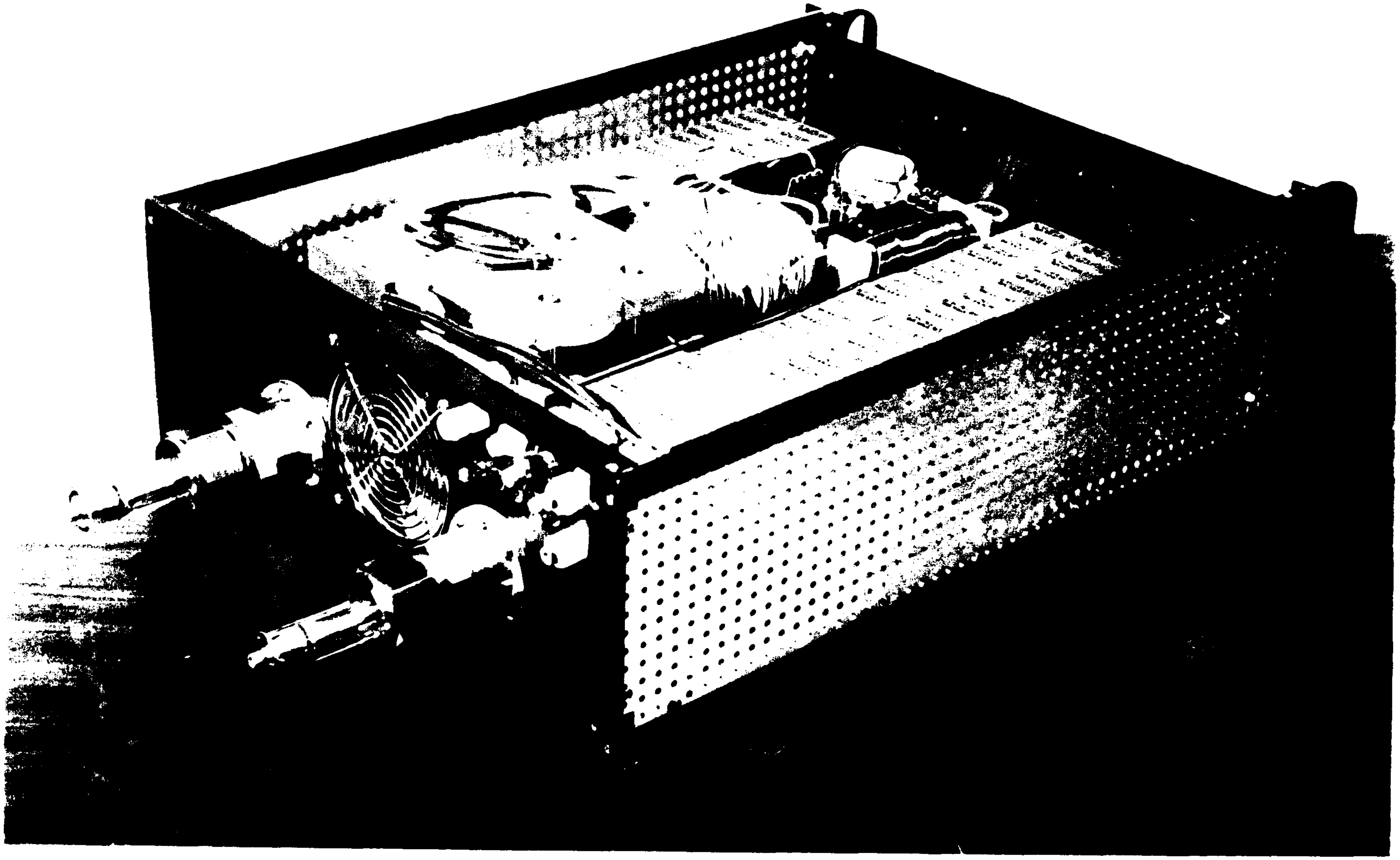


Figure 8: SSLAM-8 module, capable of producing 5 J, 12 KV pulses into a matched load at repetition rates of up to 5 KHz.

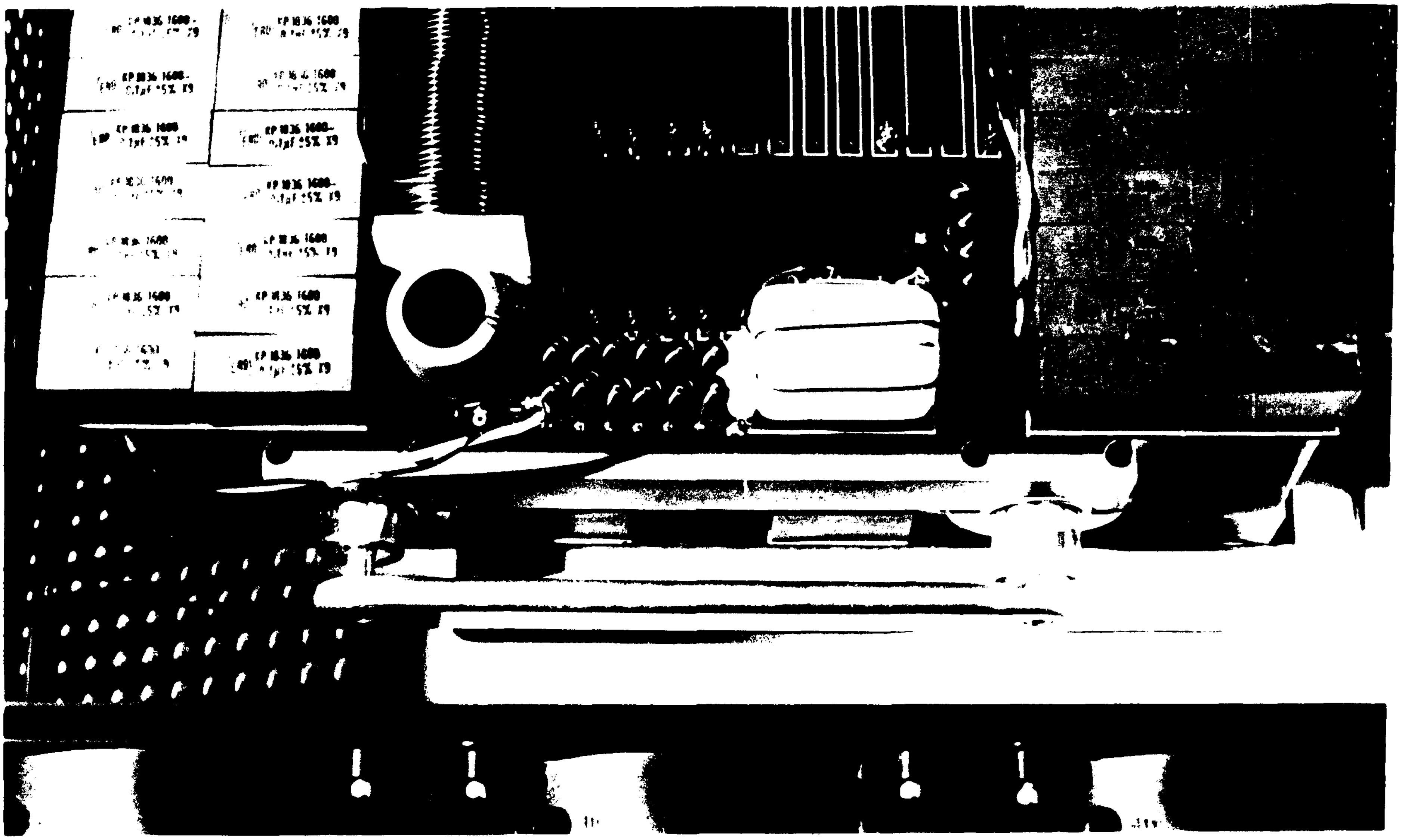


Figure 9: SSLAM-8 module showing how the isolated induction rods are strapped together to sum up the total 72 KV of output voltage.

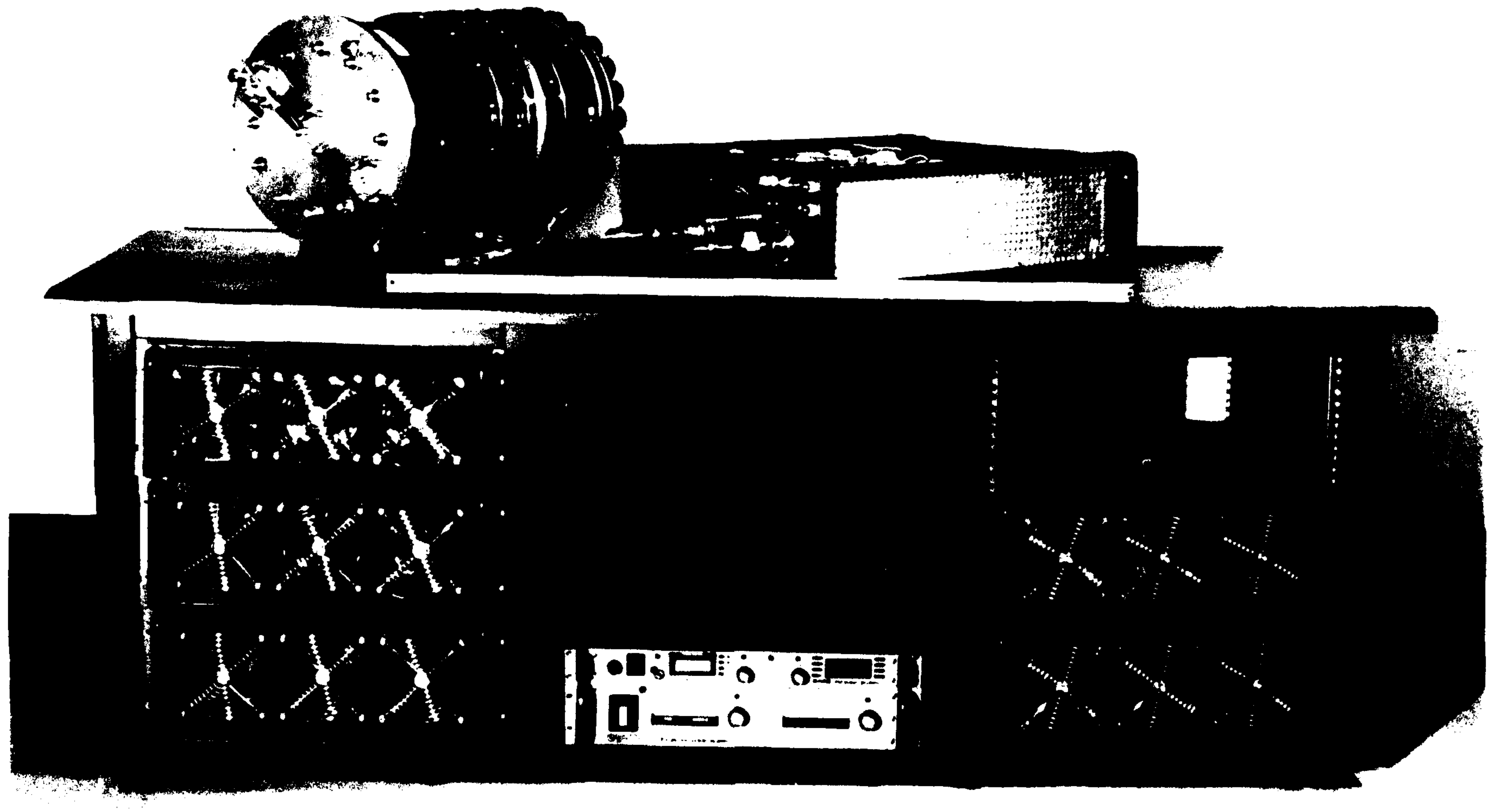


Figure 10: **Full SSLAM-8 pulsed power system with 6 modules producing 72 KV, 30 J pulses at up to 5 KHz rates. The COLD- system, shown on top, produces 30 J pulses at 10 Hz.**

SCIENCE RESEARCH LABORATORY

The ends of the tube are fitted into cylindrical electrodes which also serve to retain ZnSe windows to seal off the system. The SSLAM-VIII power supply modules were connected to supply a potential of +35 kV to the electrode at one end of the discharge and -35 kV to the electrode at the other end. A grounded ring near the electrodes at the ends of the cavity, but outside of the glass tube, served to break down the gas near the electrodes at the beginning of the pulse and thus initiate the full discharge.

The laser gas mixture was fed into the laser cavity through the electrodes at one end of the cavity and extracted through a bleed valve at the other end. Generally the laser mixture was not circulated in these tests. The system was also only run in bursts of up to several thousand pulses to avoid significant heating the walls of the discharge tube.

3.3 Conduction Cooling in a Cylindrical Geometry

Since the original design$^{(2)}$ had considered a slab geometry it is worthwhile to recalculate the effects of conduction cooling for this cylindrical geometry. The general steady-state equation for the temperature T in a gas of conductivity κ heated at a rate Q (power per unit volume) is

$$\nabla \cdot (\kappa \nabla T) = -Q \tag{1}$$

In general, both Q and κ (which is temperature dependent) can be functions of position in the gas.

In a cylindrically symmetric geometry, Eq. (1) reduces to

$$\frac{1}{r}\frac{d}{dr}\left(\kappa r \frac{dT}{dr}\right) = -Q \tag{2}$$

where r is the radial position ($0 \leq r \leq R$).

Eq. (2) can be solved numerically for a given heating-rate distribution and temperature-dependent conductivity. A simplification occurs if both Q and κ are independent of r; for this case an analytical solution may be obtained immediately. One has

$$\frac{dT}{dr} = -\frac{Q}{\kappa}\frac{r}{2}$$

Let $T_w = T(R)$, the wall temperature. Then

$$T(r) - T_w = \frac{Q}{4\kappa}\left(R^2 - r^2\right) \tag{3}$$

and thus the maximum temperature obtained, $T_{max} = T(0)$ is

$$T_{max} = T_w + \frac{Q}{16\kappa}D^2 \tag{4}$$

where $D = 2R$ is the inner diameter of the tube.

It is interesting to compare this result with the maximum temperature in a cell having a rectangular cross section of dimensions $d \times d'$ in the limit $d << d'$. In that limit the conduction to the walls separated by d dominates, and the conduction to the walls separated by d' may be neglected. For the same heating rate Q the result, easily obtained from Eq. (1) is then

$$T_{max} = T_w + \frac{Q}{8\kappa}d^2 \tag{5}$$

Thus as far as maximum temperature increase is concerned, a tube of circular cross section and diameter D is equivalent to a tube of rectangular cross section and small-wall separation $D/\sqrt{2}$. This result is reasonable: the cylindrical tube has more nearby surface area through which the heat can flow than a rectangular-cross-sectional tube with the same wall separation.

Since the volume of the heated cylinder is

$$V = \frac{\pi}{4}D^2L$$

where L is the length of the cylinder, the maximum temperature rise of the cylinder may be expressed in terms of the total heating rate

$$Q_T = VQ = \pi R^2 LQ \tag{6}$$

as follows:

$$T_{max} - T_w = \frac{Q_T}{4\pi\kappa L} \tag{7}$$

In addition, the total heating rate may be related to the laser power P and efficiency ϵ (defined as the ratio of the optical power extracted to the electrical power pumping the laser gas) by

$$Q_T = \frac{(1-\epsilon)P}{\epsilon} \tag{8}$$

so that

$$T_{max} - T_w = \frac{(1-\epsilon)P}{4\pi\epsilon\kappa L} \tag{9}$$

For a 3:2:1 He:N_2:CO_2 mixture typical of a CO_2 laser, the conductivity is $\sim 1 \times 10^{-3}$ W/cm °K. Thus for a cylindrical tube of length L, laser power P and efficiency ϵ the steady-state temperature rise at the center is

$$T_{max} - T_w \simeq 80\frac{(1-\epsilon)P}{\epsilon L} \tag{10}$$

where P is measured in watts and L is measured in cm.

Consider for example a CO_2 laser of length $L = 100$ cm, average power $P = 20$ W and efficiency $\epsilon = 10\%$. According to Eq. (10), the maximum time-averaged temperature rise is

$$\Delta T \simeq 180°\mathrm{K}$$

Temperatures on this order, of course, will drop the gas density on the longitudinal axis to about 0.6 of the gas density at the wall. The discharge will then preferentially pump the laser mixture on the axis. This effect was observed in the experiments described below.

3.4 Optimization of the Laser Output Power

Figures 11 to 14 show the voltage and current through the longitudinal discharge at pulse repetition rates of from 20 Hz to 100 Hz. In all of these particular cases the laser gas was 1:1:8 mixture of CO_2:N_2:He and only the data for the 30th pulse in a burst of 30 is shown.

The voltages were measured with an optically isolated probe consisting of an LED transmitter attached between the two discharge electrodes in series with a 2 MΩ current limiting resistor. The light output from the LED is proportional to the current through the LED and thus to the voltage across the discharge. The LED was coupled through an insulating optical fiber to a fast photodiode receiver. The entire probe was calibrated with fast, high voltage pulses from a known source. The current was measured with a fast Pearson current transformer which is also

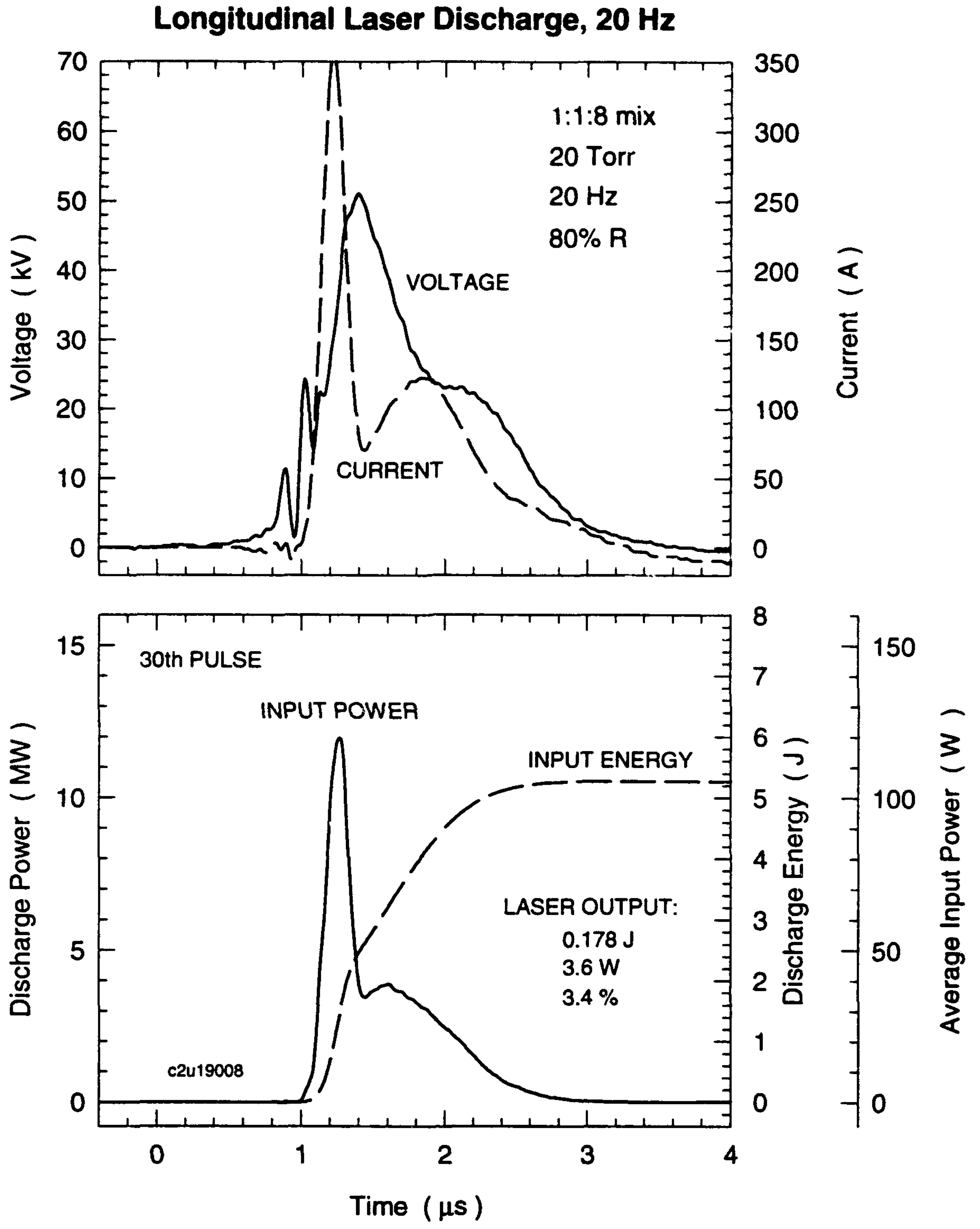


Figure 11: Voltage and current waveforms for a typical laser gas mixture and pressure at a repetition rate of 20 Hz.

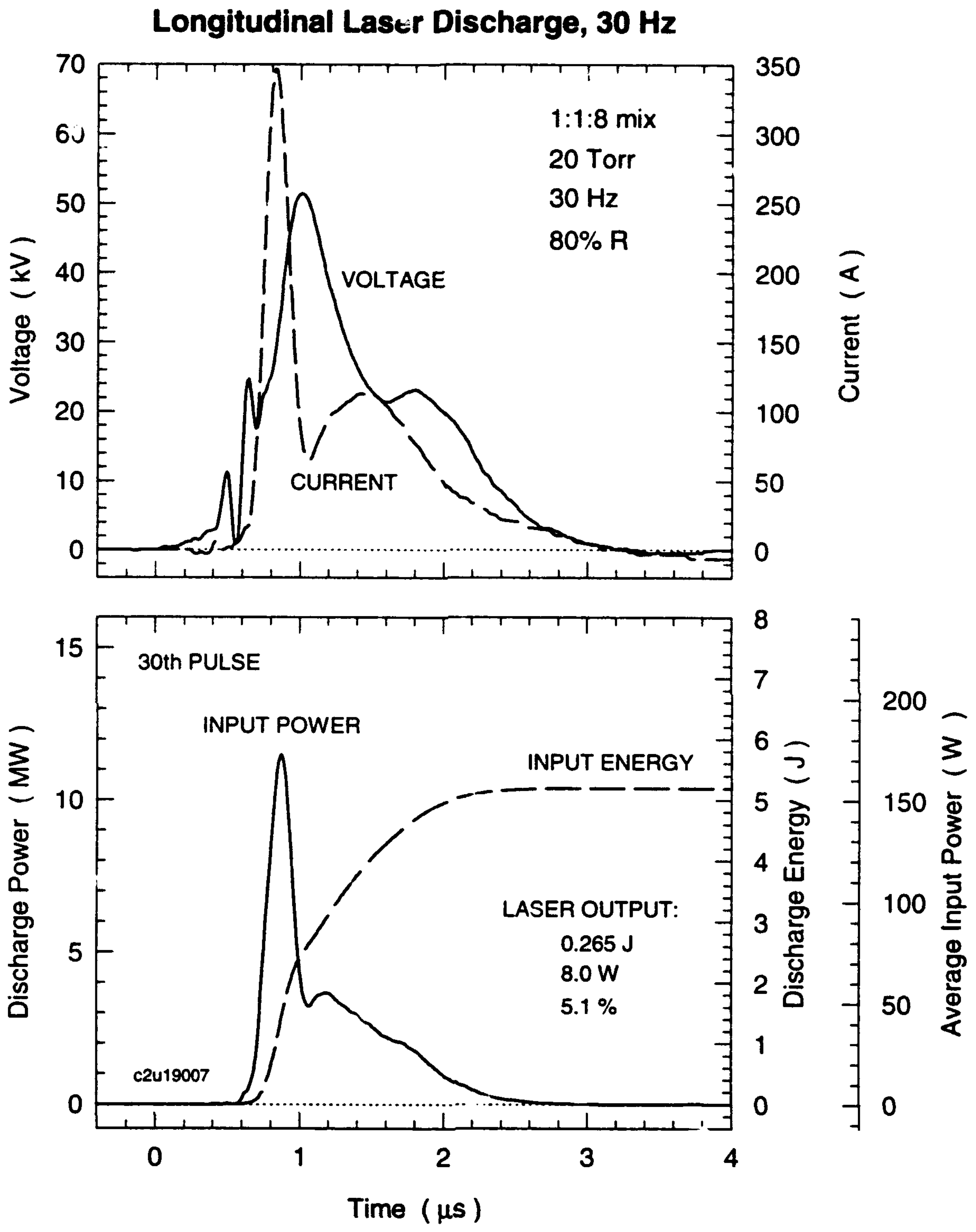


Figure 12: Voltage and current waveforms for a typical laser gas mixture and pressure at a repetition rate of 30 Hz.

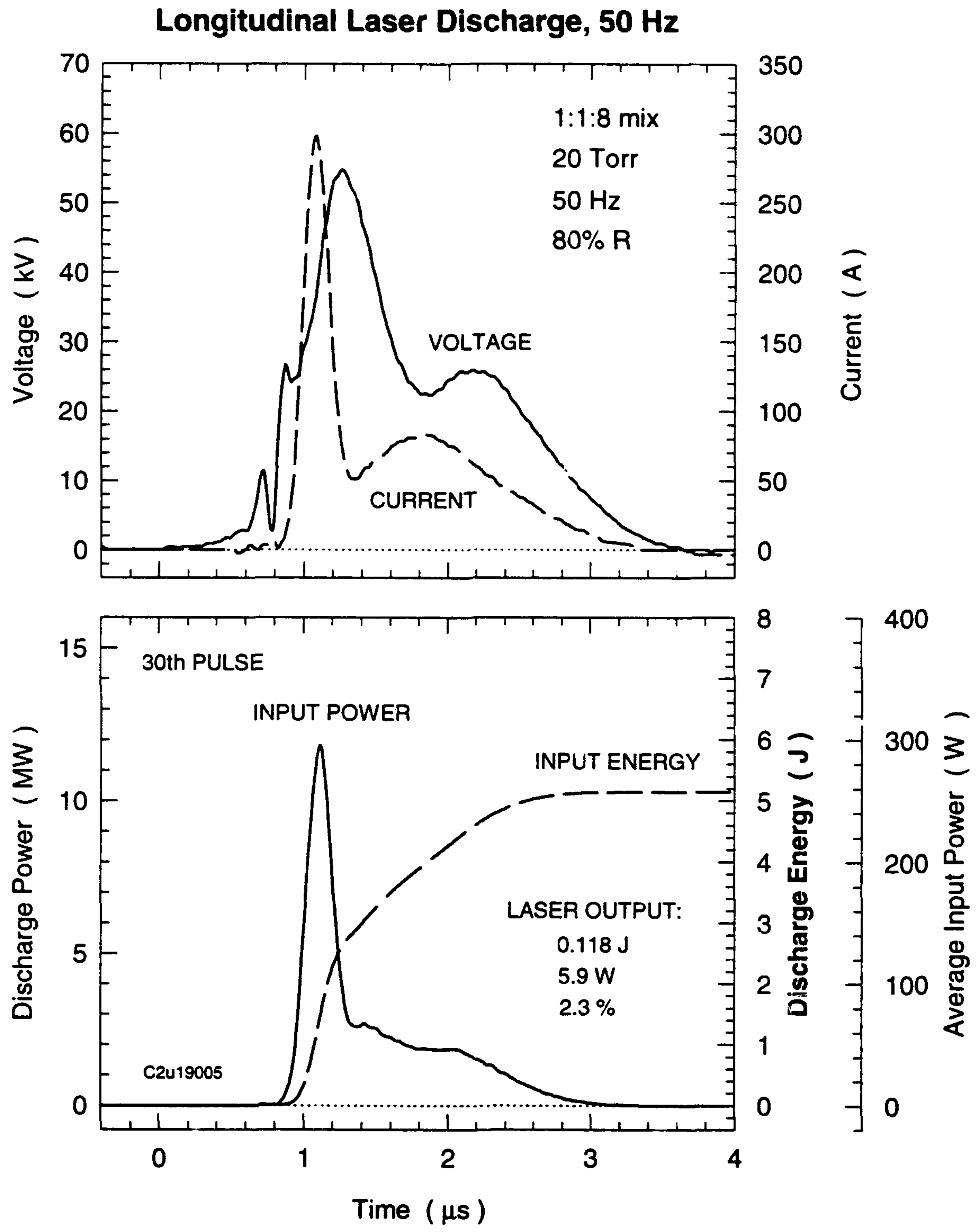


Figure 13: Voltage and current waveforms for a typical laser gas mixture and pressure at a repetition rate of 50 Hz.

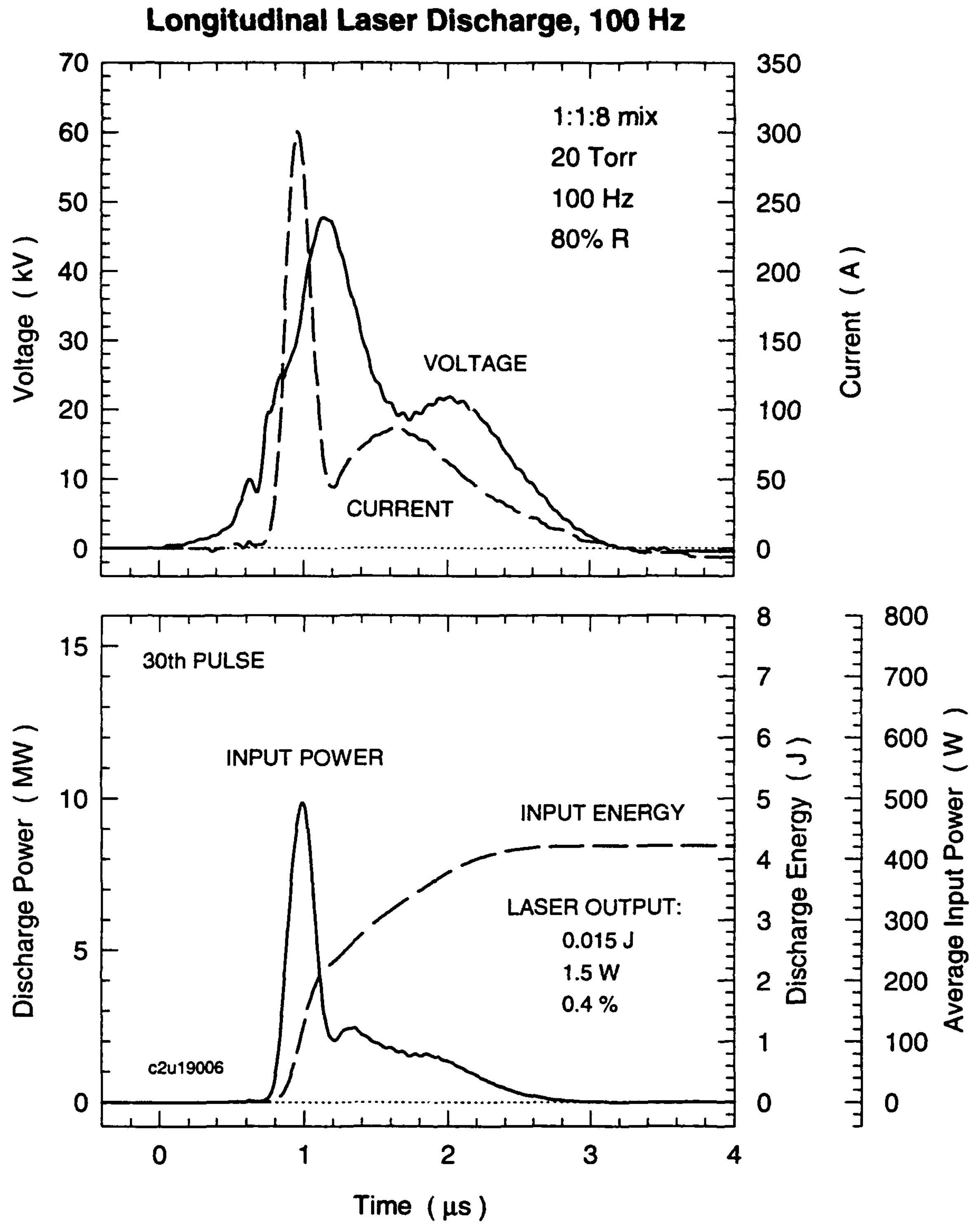


Figure 14: Voltage and current waveforms for a typical laser gas mixture and pressure at a repetition rate of 100 Hz.

isolated from the discharge circuit. Both the current and voltage waveforms were digitized at 100 MSamples/s with a Lecroy oscilloscope and stored for later processing. The input power and input energies shown on the lower graphs are calculated from these voltage and current waveforms. A separate digital scope stored the energy of the laser pulse as measured by an integrating pyroelectric joulemeter.

A total of 31 successive pulse waveforms and energies were stored for each run, starting with the first trigger to the SSLAM-VIII pulser. The first trigger pulse does not result in a discharge since the SSLAM-VIII system uses a resonant charging system. The first pulse is thus only at half voltage and the gas does not break down. The wall temperatures before each run were nominally at room temperature and the walls were sufficiently thick that they did not heat significantly during the run.

At low repetition rates the gas cools sufficiently for each pulse to be independent, except for a slow energy decay due to the dissociation of some of the CO_2. Figure 15 shows the energy for each pulse in the 30 pulse series at the repetition rates illustrated above. At higher repetition rates the behavior of the first few pulses is generally erratic, exhibiting higher energies but sometimes skipping pulses.

In all of these cases the voltage typically rises to 20 to 30 kV before the discharge is initiated. The current spikes for about 150 ns and then falls back to a more or less steady state value for about 1 μs. About half of the energy transferred to the discharge occurs during the initial voltage spike when the discharge impedance is below 100 Ω. The impedance rises following the initial current spike to about 1000 Ω and then drops again to around 200 Ω for most of the rest of the discharge. Tests on the SSLAM-VIII pulser show that it is best matched to much lower impedances of about 4 Ω. This is the reason that the total energies transferred to the discharge are typically only about 5 to 6 J rather than the 30 J that the pulser can deliver to a matched load. It is likely that the overall efficiency of the system could be improved by effectively pulse charging a local energy storage capacitor attached to the discharge tube. The

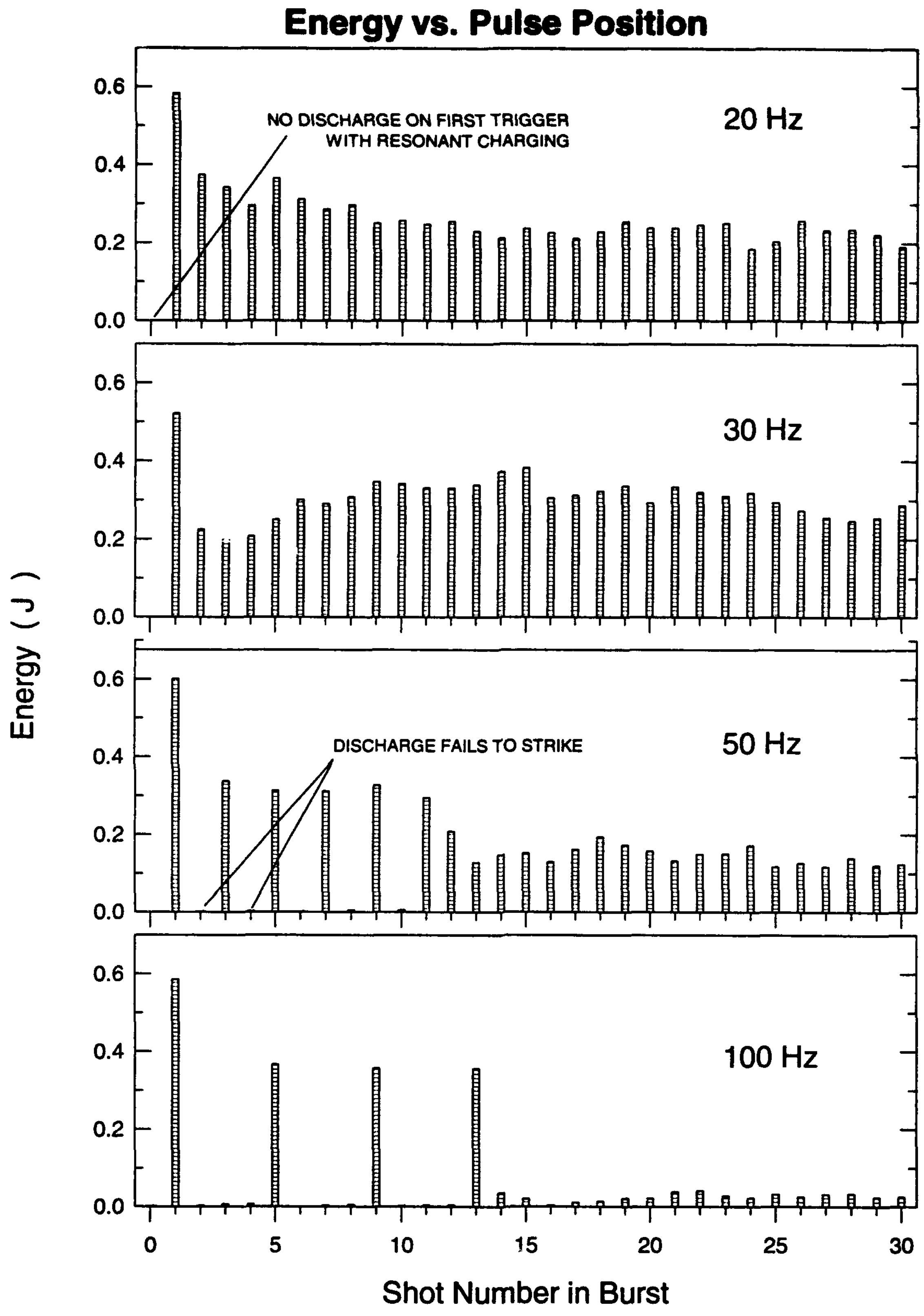


Figure 15: Energies for each laser pulse resulting from 31 successive triggers to the SSLAM-VIII pulsed power system.

charging would occur through a string of high voltage diodes which would block the return current when the SSLAM-VIII voltage pulse drops back to zero. We were not able to implement this fix before the end of the contract, however.

The voltage and current traces are essentially independent of the repetition rate of the discharge. This is not the case for the laser energy. Pulse energies peak at the 30 Hz repetition rate as shown in Fig. 15. At repetition rates above 20 Hz the discharge appears to stick close to the walls of the glass tube for the first few shots and then concentrate near the center of the tube after the gas has heated up. Above 30 Hz the steady-state pulse energies drop rapidly, presumably due to heating, so that the total laser power also peaks at 30 Hz. The maximum average power observed from this gas mixture was 8 W. Other mixtures gave peak average powers of up to 18 W at 50 Hz.

Figure 16 shows the effects of varying the gas fill pressure at the constant repetition rate of 30 Hz. The energy transfer to the discharge peaks at 25 Hz although the laser pulse energy continues to increase up the maximum fill pressure of 35 Torr. Above 35 Torr the discharge was erratic with high voltage breakdown occurring at the output terminals of the pulsed power supply.

Additives to the gas mixture, such as Xe, were tried, without much success, to see if the impedance of the discharge could be significantly lowered. Figure 17 shows the effect of adding water vapor to the gas mixture (which, in this case included 1% of Xe). The water vapor does decrease the fluctuations in the laser output energy and decrease the slow rate of energy decay observed.

Energy vs. Gas Fill Pressure

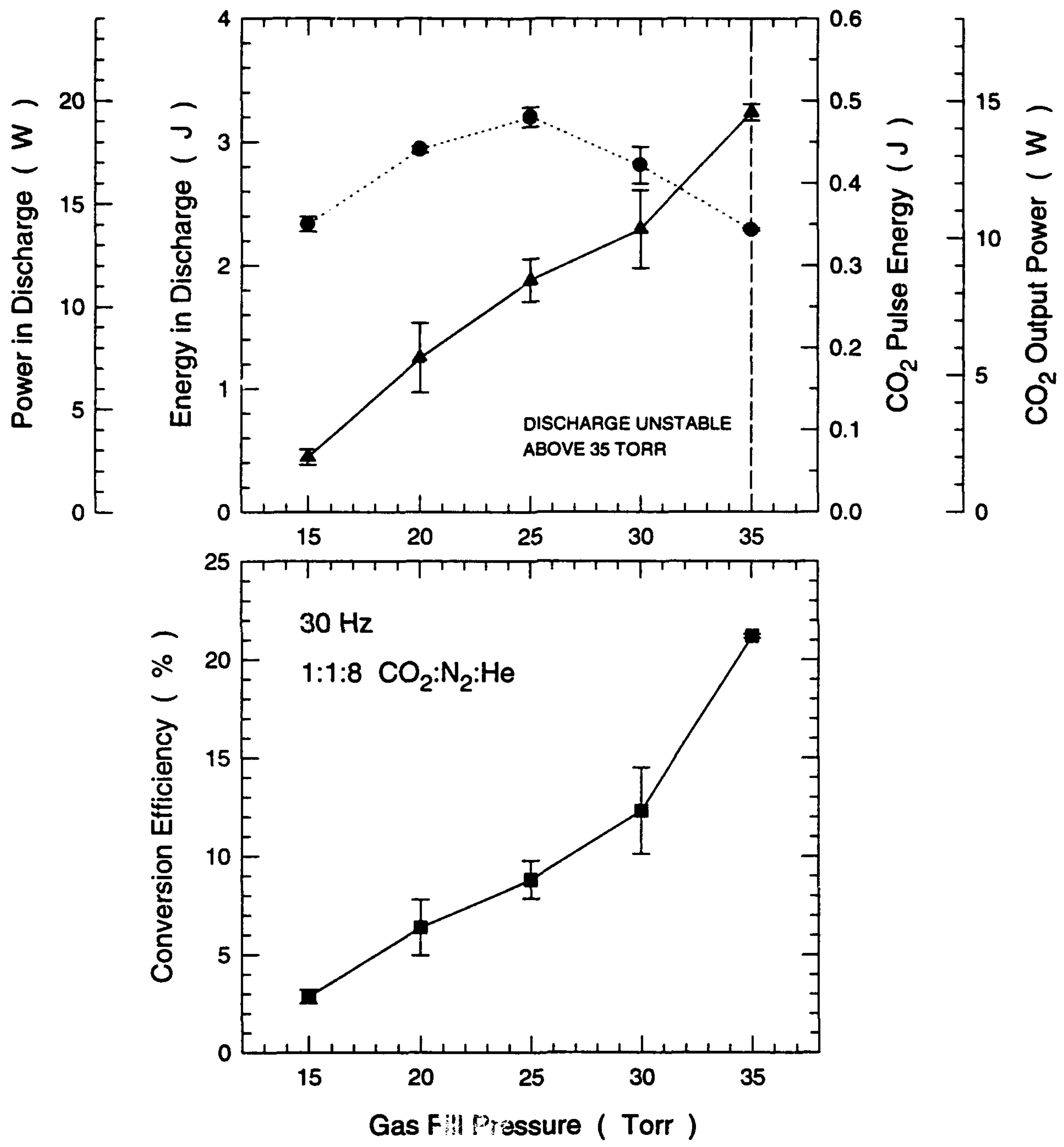


Figure 16: The laser efficiency and output power rises with gas pressure until the discharge fails to strike.

Effect of Water Vapor on Pulse Energy

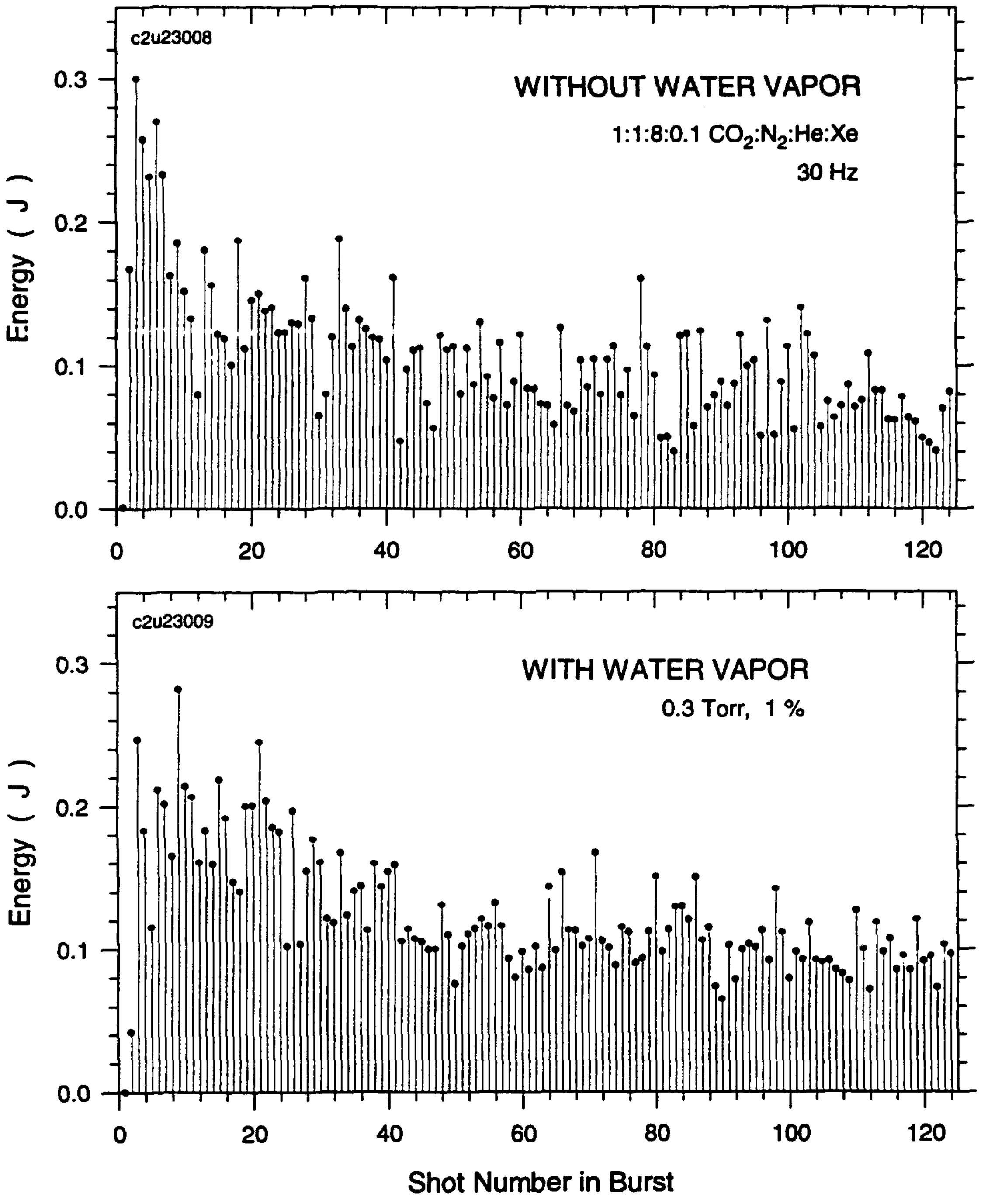


Figure 17: Adding water vapor to the discharge decreases the slow rate of the decay in pulse energy.

REFERENCES

1. J. Jacob, S. Fulghum, and H. Manning, "New Discharge Pumping Method for CO_2 Lasers", Phase II Final Report on Contract #DAAH01-88-C-0138

2. J. Jacob, "Compact Lightweight CO_2 Laser for SDIO Application", Phase II proposal submitted August 15, 1989.

APPENDIX

```
#define NFRINGE 1200
#define NGRID 41
#define NPASS 10
#define NOPTIMIZE 8
#define NSMOOTH 2
#define NPOINTS 22
#define ZFRAC (float)0.2
#define TOL (float)5.0e-5

/* GridFitZ.c  V1.0  22May92  S.F.Fulghum  Science Research Laboratory

  This version utilizes a true minimization of the surface rather
  than an average.

  This version subtracts the polynomial fit of piston, tip and tilt
  from the INPUT DATA, BEFORE proceeding with the minimization fit
  and initializes the grid with the polynomial fit without
  piston,tip,tilt.

  Calculates a wavefront from standard Mach-Zehnder fringe data
  over an NGRID x NGRID grid using a minimum surface area constraint
  to define the values at grid points not fixed by data.

  Fringe data is stored in the structured array f[i] of type fringe
  which contains the fringe x,y,z data, the location of the grid
  in which the fringe point is found and the interpolation
  weights which determine the surface value in terms of the values
  at the four corners of the grid, LL,LR,UL,UR;

  Grid data is stored in the 2-D array g[row][col] of type grid
  and is accessed through pointers to each row of the array
  in the style that Numerical Recipes uses for arrays.

  g[row][col] contains the x,y and z values of the grid point,
  a Boolean type as to whether or not the grid point is inside the
  aperture of the data (i.e. for round or rectangular apertures),
  the number of fringe points affected by changes in this grid value
  and a structured array (type gridfringe) with information about
  the fringe points affected by this grid point.

  The type gridfringe has two elements, the first of which is
  the integer index of the fringe data point in question and
  the second of which is the weight the fringe point in question
  plays in the determination of the grid point value.

  In this program rows increase from the bottom of the aperture
  to the top of the aperture and
  columns increase from the left to right so that
  col=0 -> x=-1 to col=NGRID-1 -> x=+1;
  row=0 -> y=-1 to row=NGRID-1 -> y=+1;

  INPUT FILES:

    .NFG is the normalized fringe data output from READFNG
    .POL is the polynomial fit coefficients from POLYFIT

  OUTPUT FILES:

    .GPF is the polynomial coefficients (without piston, tip, and tilt)
  as calculated at the grid positions.
    .GPT is the polynomial coefficients including tip and tilt
  evaluated at the grid positions.
    .GMF is the minimization fit to the original data (from which
  the original tip and tilt had been subtracted).  Note that
  any additional tip and tilt resulting from the minimization
  remains in this data. (possibly should be changed)
    .GMT is the minimization fit with the original tip and tilt
  added back in so that fringes can be calculated.
    .GMD is a difference filt between the .GMT file and
  a reference .GMT file.


  The .GMT output file contains the minimization fit (to data from which
  piston, tip and tilt had been subtracted) with the original tip and
  tilt put back in.  If GRIDFITS is run on a null shot, it will generate
  a .GMT file suitable for use as a reference surface.

  The .GMD file is the DIFFERENCE between a .GMT file and a REFERENCE .GMT
  file so that an added aberration can be dstermined which INCLUDES
  any added tip and tilt.
```

```
*/
/* ******************************************************************* */
#define TRUE 1
#define FALSE 0
#include <stdio.h>
#include <process.h>
#include <malloc.h>
#include <math.h>
#include <string.h>

/* ******************************************************************* */
/* function declarations */

float poly(int m, float x, float y);
float polyfit(float *pcoef, float x, float y);
float RMSfit(void);
void move_grid_point(void);
void smooth_grid(void);
void optimize_grid(void);
float L(float z);
void zlimits(float *zmin, float *zmax, float *zavg);
void minimize_surface_tension(void);
float sfbrent(float ax, float bx, float cx, float (*f)(float x),
      float tol, float *xmin, int *niter);

/* ******************************************************************* */
/* Global Variables

  These structures contain the input fringe positions and orders,
  the grid values, the information as to which fringe points are
  affected by each grid value and the constants required for
  interpolating the surface between the grid points at the positions
  of each of the input fringe data points.

*/

  struct fringe
  {
    float x;   /* x value of fringe point */
    float y;   /* y value of fringe point */
    float z;   /* order of fringe (integral but float for calculation) */
    int rLL;   /* row of LL corner of grid square containing the point */
    int cLL;   /* column of lower left corner */
    float wLL; /* weight of LL grid value for the x,y value of the point */
    float wLR;
    float wUL;
    float wUR;
  };

  struct grid
  {
    float xg;   /* x value at the grid point */
    float yg;   /* y value at the grid point */
    float zg;   /* z value at the grid point (the final output) */
    int   in;   /* 1 if inside the aperture, 0 if outside */
    int   nf;   /* number of fringe points affecting grid point */
    int   fi[ NPOINTS ]; /* vector of fringe indices */
  };

  struct fringe *f;
  int npts;
  struct grid  **g;
  float **temp;
  int row,col;

/* ******************************************************************* */
void main()
  {

  FILE *fp;
  char basename[40],name[40],infile[80],polyfile[80],polyfitfile[80];
  char polytipfile[80],minfitfile[80],mintipfile[80];
  char subfile[80],diffile[80];
  int i,j,rLL,cLL,nf,ifp,maxnf;
  int ipass,iopt,ismooth;
  size_t vecsize;
  float delta,xg,yg,xmin,ymin,xmax,ymax,frLL,fcLL,u,v,wfp;
  float zero=(float)0;
  float two=(float)2.0;
  float one=(float)1.0;
  float sum,diff,pcoef[16];
  float rms,rmspol,rms0;
  float avgdiff0,avgdiff;
  float zLL,zLR,zUL,zUR;
  float wLL,wLR,wUL,wUR;
  float gsub;

/* ------------------------------------------------------------------ */
/* get filenames */

  printf("enter normalized fringe filename, d:\\fringe\\dat\\*.NFG assumed\n");
  strcpy(basename,"d:\\fringe\\dat\\");
  gets(name);
  strcat(basename,name);
  printf("\n");

/* input data file of normalized fringes */
  strcpy(infile,basename);
  strcat(infile,".NFG");
  printf("infile:      %s\n",infile);

/* input data file of polynomial fit coefficients */
  strcpy(polyfile,basename);
  strcat(polyfile,".POL");
  printf("polyfile:    %s\n",polyfile);

/* output data file of initial polynomial fit,    */
/* WITHOUT the piston, tip and tilt coefficients */
  strcpy(polyfitfile,basename);
  strcat(polyfitfile,".GPF");
  printf("polyfitfile: %s\n",polyfitfile);

/* output data file of initial polynomial fit, WITH original tip and tilt */
 strcpy(polytipfile,basename);
  strcat(polytipfile,".GPT");
  printf("polytipfile: %s\n",polytipfile);

/* output data file of minimization fit, WITHOUT original tip and tilt */
  strcpy(minfitfile,basename);
  strcat(minfitfile,".GMF");
  printf("minfitfile:  %s\n",minfitfile);

/* output data file of minimization fit, WITH original tip and tilt */
/* for plotting contours to check against input fringes */
  strcpy(mintipfile,basename);
  strcat(mintipfile,".GMT");
  printf("mintipfile:  %s\n",mintipfile);

/* output data file of difference between two minimization fits */
/* without tip and tilt on either */
  strcpy(diffile,basename);
  strcat(diffile,".GMD");
  printf("diffile:     %s\n",diffile);

/* input data file of minimization fit to SUBTRACT from .GMT*/
/* including tip and tilt */
  printf("enter subtraction filename, .GMT assumed\n");
/*  gets(basename); */
  strcpy(basename,"d:\\fringe\\dat\\cln05024");
  strcpy(subfile,basename);
  strcat(subfile,".GMT");
  printf("subfile:     %s\n\n",subfile);

  printf("hit key to continue\n");
  getchar();


/* ------------------------------------------------------------------ */
/* allocate memory

  NFRINGE is the number of fringe points that can be stored (typ. 1000).
  NGRID is the size of the grid, an odd value to insure a grid point
  at the exact center of the grid (typ. 41).
  NPOINTS is the maximum number of input fringe points that can be
  assigned to a given grid point. The value required will vary with
  the density of sampling the input fringes and the density of grid
  points (typ. 10).

*/

  vecsize=(size_t) NFRINGE *sizeof(struct fringe);
  f=(struct fringe *)malloc(vecsize);
  if (f==NULL) {
    printf("error allocating space for fringe data\n");
    exit(1);
  }

  vecsize=(size_t) NGRID *sizeof(struct grid *);
  g=(struct grid **)malloc(vecsize);
  if (g==NULL) {
    printf("error allocating space for grid row pointers\n");
    exit(1);
  }
  vecsize=(size_t) NGRID *sizeof(struct grid);
  for(i=0;i< NGRID ;i++) {
    g[i]=(struct grid *)malloc(vecsize);
    if (g[i]==NULL) {
      printf("error allocating space for grid row %d data\n",i);
      exit(1);
    }
  }

  vecsize=(size_t) NGRID *sizeof(float *);
  temp=(float **)malloc(vecsize);
  if (temp==NULL) {
    printf("error allocating space for temp grid\n");
    exit(1);
  }
  vecsize=(size_t) NGRID *sizeof(float);
  for(i=0;i< NGRID ;i++) {
    temp[i]=(float *)malloc(vecsize);
    if (temp[i]==NULL) {
      printf("error allocating space for temp grid row %d\n",i);
      exit(1);
    }
  }

/* ------------------------------------------------------------------ */
/* initialize fringe and grid arrays */

  for(i=0;i< NFRINGE ;i++) {
    f[i].x=zero;
    f[i].y=zero;
    f[i].z=zero;
    f[i].rLL=0;
    f[i].cLL=0;
    f[i].wLL=zero;
    f[i].wLR=zero;
    f[i].wUL=zero;
    f[i].wUR=zero;
  }

  delta = two/((float)NGRID -one);
  yg=(float)-1;
  for(row=0;row< NGRID ;row++) {
    xg=(float)-1;
    for(col=0;col< NGRID ;col++) {
      g[row][col].xg=xg;
      g[row][col].yg=yg;
      g[row][col].zg=zero;
      g[row][col].in= TRUE ;
      g[row][col].nf=0;
      for(i=0;i< NPOINTS ;i++) g[row][col].fi[i]=0;
      xg+=delta;
    }
    yg+=delta;
  }

/* ------------------------------------------------------------------ */
/* read polynomial coefficients */

  printf("reading file of polynomial coefficients\n");
  fp=fopen(polyfile,"r");
  if(fp==NULL) {
    printf("error opening polynomial file: %s\n",polyfile);
    exit(1);
  }
  for(i=1;i<=15;i++) {
    fscanf(fp," %f",&(pcoef[i]));
    printf("%d  %g\n",i,pcoef[i]);
  }
  fclose(fp);
```

```
/* ------------------------------------------------------------------ */
/* read fringe data
  This section reads an arbitrarily long list of data triplets:
  x-position  y-position  fringe order
*/

  printf("reading fringe data\n");
  fp=fopen(infile,"r");
  if(fp==NULL) {
    printf("error opening fringe data file: %s\n",infile);
    exit(1);
  }
  i=0;
  while(!feof(fp)) {
    fscanf(fp," %f %f %f",&(f[i].x),&(f[i].y),&(f[i].z));
    i++;
  }
  npts=i-1;
  if (ferror(fp)) {
    printf("error reading fringe data file\n");
    fclose(fp);
    exit(1);
  }
  fclose(fp);
  printf("%d fringe points read from file\n",npts);

/* ------------------------------------------------------------------ */
/* subtract piston, tip, tilt from fringe data
   pcoef[1] is constant, pcoef[2] is x, pcoef[3] is y
*/

  printf("subtracting piston, tip, tilt from fringe data\n");
  for(i=0;i<npts;i++) {
    sum=(float)0;
    for(j=1;j<=3;j++) {
      sum+=pcoef[j]*poly(j,f[i].x,f[i].y);
    }
    f[i].z-=sum;
  }

/* ------------------------------------------------------------------ */
/* Initialize grid with polynomial fit
   WITHOUT piston, tip and tilt.
*/

  for(row=0;row< NGRID ;row++) {
    for(col=0;col< NGRID ;col++) {
      sum=zero;
      for(i=4;i<=15;i++) {
 sum+=pcoef[i]*poly(i,g[row][col].xg,g[row][col].yg);
      }
      g[row][col].zg=sum;
    }
  }

/* ------------------------------------------------------------------ */
/* write initial polynomial grid data to .GPT file
   and the fit WITHOUT piston, tip and tilt to .GPF file
   Uses exponential notation to comply with Mathematica
   for plotting results.
*/

  fp=fopen(polyfitfile,"w");
  if(fp==NULL) {
    printf("error opening grid polynomial file: %s\n",polyfitfile);
    exit(1);
  }
  for(row=0;row< NGRID ;row ++) {
    for(col=0;col< NGRID ;col++) {
      sum=zero;
      for(i=4;i<=15;i++)
 sum+=pcoef[i]*poly(i,g[row][col].xg,g[row][col].yg);
      fprintf(fp,"%-11.3e\n",sum);
    }
  }
  if (ferror(fp)) {
    printf("error writing grid polynomial fit file\n");
    fclose(fp);
    exit(1);
  }
  fclose(fp);
  fp=fopen(polytipfile,"w");
  if(fp==NULL) {
    printf("error opening grid polynomial file: %s\n",polytipfile);
    exit(1);
  }
  for(row=0;row< NGRID ;row ++) {
    for(col=0;col< NGRID ;col++) {
      sum=zero;
      for(i=1;i<=15;i++)
 sum+=pcoef[i]*poly(i,g[row][col].xg,g[row][col].yg);
      fprintf(fp,"%-11.3e\n",sum);
    }
  }
  if (ferror(fp)) {
    printf("error writing grid polynomial fit file\n");
    fclose(fp);
    exit(1);
  }
  fclose(fp);

/* ----------------------------------------------------------------------- */
/* distribute fringe data points among the grid points

  This section looks at each of the input fringe data points
  and determines which four grid points surround it.
  It then writes this information into the data structures for
  each of these four grid points along with the interpolation
  constants (weights) for the fringe point.

  The interpolation scheme used for the surface between four
  grid points, lower left LL, lower right LR, upper left UL
  and upper right UR is as follows.  The fractional distance
  of a fringe data point position from the left edge of the surface
  towards the right edge of the surface is "u". The fractional
  distance from the bottom edge to the top edge of the surface is "v".
  The suface value "z" is defined by
  z= z_LL(1-u)(1-v) +z_UL(1-u)(v) +z_LR(u)(1-v) +z_UR(u)(v).
  This surface is continuous for all values of the grid points
  but its derivative is not.  For a point on the line connecting two
  grid points the interpolation reduces to simply a linear interpolation
  between only those two points.  Note that the surface is not flat
  so the fringes that it predicts are curved within a surface.

  The values "u" and "v" are determined by calculating the float
  value of the grid position of the fringe point (i.e. row=2.15 col=20.45)
  and subtracting the integral portion.  The integral portion is used to
  determine the index of the lower left LL grid point of the surface
  in which the fringe point is found.  The value (1-u)(1-v) is, for
  example, the weight of the lower left grid point in determining the
  value of the surface.  It is also the weight of the fringe point when
  determining the optimum value for the lower left grid point that
  minimizes the error between the fringe predicted by the surface and the
  measured fringe position.  If the fringe point is very close to the
  upper right corner of the surface (u approx. 1)(v approx 1) then
  it is primarily affected by the upper right grid point
  and not the lower left grid point.

*/

  for(i=0;i<npts;i++) {
    fcLL=(f[i].x+one)/delta;  /* fringe position in grid units */
    cLL=(int)fcLL;                 /* integral porion of the position */
    f[i].cLL=cLL;                  /* write this to fringe structure */
    frLL=(f[i].y+one)/delta;
    rLL=(int)frLL;
    f[i].rLL=rLL;
    u=fcLL-(float)cLL;   /* fractional position for weighting */
    v=frLL-(float)rLL;
    f[i].wLL=(one-u)*(one-v);  /* weighting values */
    f[i].wUR=u*v;
    f[i].wUL=(one-u)*v;
    f[i].wLR=u*(one-v);

    row=rLL;  /* write data to LL grid point data structure */
    col=cLL;
    wfp=f[i].wLL;
    if((row>=0)&&(row< NGRID )&&(col>=0)&&(col< NGRID )) {
      nf=g[row][col].nf;            /* fringe point counter starts at 0 */
      if(nf< NPOINTS ) {           /* index of fringe point affected */
 g[row][col].fi[nf]=i;
```

```
      }
      nf++;                          /* increment fringe point counter */
      g[row][col].nf=nf;             /* update fringe point count */
    }

    row=rLL+1;  /* UR grid point */
    col=cLL+1;
    wfp=f[i].wUR;
    if((row>=0)&&(row< NGRID )&&(col>=0)&&(col< NGRID )) {
      nf=g[row][col].nf;             /* fringe point counter starts at 0 */
      if(nf< NPOINTS ) {             /* first point stored in fi[0] */
 g[row][col].fi[nf]=i;        /* index of fringe point */
      }
      nf++;                          /* increment fringe point counter */
      g[row][col].nf=nf;             /* update fringe point count */
    }

    row=rLL+1;  /* UL grid point */
    col=cLL;
    wfp=f[i].wUL;
    if((row>=0)&&(row< NGRID )&&(col>=0)&&(col< NGRID )) {
      nf=g[row][col].nf;             /* fringe point counter starts at 0 */
      if(nf< NPOINTS ) {            /* point 1 stored in fi[0] */
 g[row][col].fi[nf]=i;         /* index of fringe point */
      }
      nf++;                          /* increment fringe point counter */
      g[row][col].nf=nf;             /* update fringe point count */
    }

    row=rLL;    /* LR grid point */
    col=cLL+1;
    wfp=f[i].wLR;
    if((row>=0)&&(row< NGRID )&&(col>=0)&&(col< NGRID )) {
      nf=g[row][col].nf;             /* fringe point counter starts at 0 */
      if(nf< NPOINTS ) {            /* point 1 stored in fi[0] */
 g[row][col].fi[nf]=i;        /* index of fringe point */
      }
      nf++;                          /* increment fringe point counter */
      g[row][col].nf=nf;             /* update fringe point count */
    }

  } /* end of fringe point loop */
  printf("%d fringe points distributed\n",npts);

  maxnf=0;
  for(row=0;row<NGRID;row++) {
    for(col=0;col<NGRID;col++) {
      if(g[row][col].nf>maxnf) maxnf=g[row][col].nf;
    }
  }
  printf("maxnf=%d, space allocated for %d\n",maxnf, (int)NPOINTS );
  if(maxnf>= NPOINTS ) {
    printf("too little space allocated for fringes\n");
    exit(1);
  }

/* ------------------------------------------------------------------ */
/* calculate initial fit error */

  rms0=RMSfit();
  printf("initial RMS fit error= %g\n",rms0);

/* ################################################################## */

  for(ipass=1;ipass<= NPASS ;ipass++) {

    printf("pass %d\n",ipass);
    for(ismooth=1;ismooth<= NSMOOTH ;ismooth++) smooth_grid();
    rms=RMSfit();
    printf("  after smooth rms error= %g\n",rms);

    for(iopt=1;iopt<= NOPTIMIZE ;iopt++) optimize_grid();
    rms=RMSfit();
    printf("   after opt rms error= %g\n\n",rms);

  } /* end optimization loops */

/* ################################################################## */

/* ------------------------------------------------------------------ */
/* calculate final fit error */
```

```
  printf("\n initial RMS fit error= %g\n",rms0);
  printf("\n final RMS fit error= %g\n",rms);

/* ------------------------------------------------------------------ */
/* write final grid data to .GMF file
   Uses exponential notation to comply with Mathematica
   for plotting results.
*/

  printf("writing grid minimization fit in .GMF file\n");
  fp=fopen(minfitfile,"w");
  if(fp==NULL) {
    printf("error opening minimization grid fit file: %s\n",minfitfile);
    exit(1);
  }
  for(row=0;row< NGRID ;row ++) {
    for(col=0;col< NGRID ;col++) {
      fprintf(fp,"%-11.3e\n",g[row][col].zg);
    }
  }
  if (ferror(fp)) {
    printf("error writing grid minimization fit file\n");
    fclose(fp);
    exit(1);
  }
  fclose(fp);

/* ------------------------------------------------------------------ */
/* write final grid data to .GMT file
   in which the polynomial piston, tip, tilt are put back in
*/

  printf("writing minimization fit with original tip and tilt in .GMT file\n");
  fp=fopen(mintipfile,"w");
  if(fp==NULL) {
    printf("error opening fit minus polynomial file: %s\n",mintipfile);
    exit(1);
  }
  for(row=0;row< NGRID ;row ++) {
    for(col=0;col< NGRID ;col++) {
      sum=zero;
      for(i=1;i<=3;i++)
 sum+=pcoef[i]*poly(i,g[row][col].xg,g[row][col].yg);
      fprintf(fp,"%-11.3e\n",(g[row][col].zg+sum));
    }
  }
  if (ferror(fp)) {
    printf("error writing fit minus polynomial file: %s\n",mintipfile);
    fclose(fp);
    exit(1);
  }
  fclose(fp);

/* ------------------------------------------------------------------ */
/* write final grid data to .GMD file
   Uses exponential notation to comply with Mathematica.
   Subtract a set of grid points first using a full file name.
*/

  printf("reading subtraction file: %s\n",subfile);
  fp=fopen(subfile,"r");
  if(fp==NULL) {
    printf("error opening subtraction file: %s\n",subfile);
    exit(1);
  }
  for(row=0;row< NGRID ;row ++) {
    for(col=0;col< NGRID ;col++) {
      fscanf(fp," %e",&(temp[row][col]));
    }
  }
  if (ferror(fp)) {
    printf("error reading subtraction file: %s\n",subfile);
    fclose(fp);
    exit(1);
  }
  fclose(fp);

  printf("writing difference file: %s\n",diffile);
  fp=fopen(diffile,"w");
  if(fp==NULL) {
    printf("error opening grid difference file: %s\n",diffile);
    exit(1);
  }
  for(row=0;row< NGRID ;row ++) {
    for(col=0;col< NGRID ;col++) {
      sum=zero;
      for(i=1;i<=3;i++)
 sum+=pcoef[i]*poly(i,g[row][col].xg,g[row][col].yg);
      fprintf(fp,"%-11.3e\n",(g[row][col].zg+sum -temp[row][col]));
    }
  }
  if (ferror(fp)) {
    printf("error writing grid difference file: %s\n",diffile);
    fclose(fp);
    exit(1);
  }
  fclose(fp);

/* ------------------------------------------------------------------- */
/* free memory space */

  for(i=0;i< NGRID ;i++) free(g[i]);
  free(g);
  for(i=0;i< NGRID ;i++) free(temp[i]);
  free(temp);
  free(f);
}
/* ***************************************************************** */
float poly(int m, float x, float y)
{
  switch(m) {
  case 1: return (float)1;
   break;
  case 2: return x;
   break;
  case 3: return y;
   break;
  case 4: return x*x;
   break;
  case 5: return x*y;
   break;
  case 6: return y*y;
   break;
  case 7: return x*x*x;
   break;
  case 8: return x*x*y;
   break;
  case 9: return x*y*y;
   break;
  case 10: return y*y*y;
    break;
  case 11: return x*x*x*x;
    break;
  case 12: return x*x*x*y;
    break;
  case 13: return x*x*y*y;
    break;
  case 14: return x*y*y*y;
    break;
  case 15: return y*y*y*y;
    break;
  default: printf("f subscript out of bounds\n");
    break;
  }
}
/* ================================================================== */
float polyfit(float *pcoef, float x, float y)
{
  int m;
  float sum;

  sum=(float)0;
  for(m=1;m<=15;m++) {
    sum+=pcoef[m]*poly(m,x,y);
  }
  return(sum);
}
/* ================================================================== */
float RMSfit(void)
{
  int i,rLL,cLL;
  double sum,zdiff,zval;

  sum=(double)0;
  for(i=0;i<npts;i++) {
    rLL=f[i].rLL;
    cLL=f[i].cLL;
    zval= (double)g[rLL  ][cLL  ].zg*(double)f[i].wLL
  +(double)g[rLL+1][cLL  ].zg*(double)f[i].wUL
  +(double)g[rLL+1][cLL+1].zg*(double)f[i].wUR
  +(double)g[rLL  ][cLL+1].zg*(double)f[i].wLR;
    zdiff=(double)f[i].z-zval;
    sum+=zdiff*zdiff;
  }
  return (float)sqrt( sum/(double)npts );
}
/* ================================================================= */
void move_grid_point(void)
{
  int i,ifp,rLL,cLL;
  float avgdiff;

  avgdiff=(float)0;
  for(i=0;i<g[row][col].nf;i++) {
    ifp=g[row][col].fi[i];     /* index of fringe point */
    rLL=f[ifp].rLL;
    cLL=f[ifp].cLL;
    avgdiff+= f[ifp].z
      -g[rLL  ][cLL  ].zg*f[ifp].wLL
      -g[rLL  ][cLL+1].zg*f[ifp].wLR
      -g[rLL+1][cLL  ].zg*f[ifp].wUL
      -g[rLL+1][cLL+1].zg*f[ifp].wUR;
  }
  avgdiff/=(float)g[row][col].nf;
  g[row][col].zg+= ZFRAC *avgdiff;
}
/* ================================================================= */
/* smooth grid */

void smooth_grid(void)
{
  float zsum,nsum;

  printf("smo ");
  for(row=0;row< NGRID; row++) {
    for(col=0; col< NGRID ;col++) {
      zsum=g[row][col].zg+g[row][col].zg;
      nsum=(float)2;
      if(row>0) {
        zsum+=g[row-1][col].zg;
        nsum+=(float)1;
      }
      if(row< NGRID -1) {
        zsum+=g[row+1][col].zg;
        nsum+=(float)1;
      }
      if(col>0) {
        zsum+=g[row][col-1].zg;
        nsum+=(float)1;
      }
      if(col< NGRID -1) {
        zsum+=g[row][col+1].zg;
        nsum+=(float)1;
      }
      temp[row][col]=zsum/nsum;
    }
  }
  for(row=0;row< NGRID; row++)
    for(col=0; col< NGRID ;col++)
      g[row][col].zg=temp[row][col];
}
/* ================================================================= */
/* optimize grid */

void optimize_grid(void)
{

  printf("opt ");
  for(row=0;row< NGRID; row++) {
    for(col=0; col< NGRID ;col++) {
      if(g[row][col].nf==0) minimize_surface_tension();
      else move_grid_point();
    }
  }
}
/* ==================================================================== */
/* Length of Grid Connections, L (actually returned as L^2) */

float L(float z)
{
  float zdif,Ltotal;

  Ltotal=(float)0;
  if(row>0) {
    zdif=z-g[row-1][col].zg;
    Ltotal+=sqrt( (float)1 +zdif*zdif );
  }
  if(row< NGRID -1) {
    zdif=z-g[row+1][col].zg;
    Ltotal+=sqrt( (float)1 +zdif*zdif );
  }
  if(col< NGRID -1) {
    zdif=z-g[row][col+1].zg;
    Ltotal+=sqrt( (float)1 +zdif*zdif );
  }
  if(col>0) {
    zdif=z-g[row][col-1].zg;
    Ltotal+=sqrt( (float)1 +zdif*zdif );
  }
  return Ltotal;
}
/* ==================================================================== */
/* Minimum and Maximum of surrounding points */

void zlimits(float *zmin, float *zmax, float *zavg)
{
  float z,zsum,nsum;

/*
  if (row==20) {
     printf("row=20\n");
  }
*/
  *zmin=(float)1.0e10;
  *zmax=(float)-1.0e10;
  zsum=(float)0;
  nsum=(float)0;
  if(row>0) {
    z=g[row-1][col].zg;
    zsum+=z;
    nsum+=(float)1;
    if (*zmax<z) *zmax=z;
    if (*zmin>z) *zmin=z;
  }
  if(row< NGRID -1) {
    z=g[row+1][col].zg;
    zsum+=z;
    nsum+=(float)1;
    if (*zmax<z) *zmax=z;
    if (*zmin>z) *zmin=z;
  }
  if(col>0) {
    z=g[row][col-1].zg;
    zsum+=z;
    nsum+=(float)1;
    if (*zmax<z) *zmax=z;
    if (*zmin>z) *zmin=z;
  }
  if(col< NGRID -1) {
    z=g[row][col+1].zg;
    zsum+=z;
    nsum+=(float)1;
    if (*zmax<z) *zmax=z;
    if (*zmin>z) *zmin=z;
  }
  if(nsum>(float)0)   *zavg=zsum/nsum;
  else {
    printf("nsum=0 logic error\n");
    exit(1);
  }
}
/* ==================================================================== */
/* Minimize surface tension */
```

```
void minimize_surface_tension(void)
{
  int niter;
  float z,zmax,zmin,zavg,znew;

  z=g[row][col].zg;
  zlimits(&zmin,&zmax,&zavg);
  sfbrent(zmin,zavg,zmax,L, TOL ,&z,&niter);
  g[row][col].zg=z;
}
```

```
/* ReadFng.c

  reads the .FNG file containing the fringe positions in pixels
  and converts them to a .NFG or Normalized FrinGe file
  consisting of simple list of triplets: x y order
  normalized to a -1 0 +1 square.

*/
/* ********************************************************************** */
#include <stdio.h>
#include <string.h>
#include <process.h>

/* ********************************************************************** */
void main()
{
  FILE *fp,*fpin,*fpout;
  char basefilename[80],name[40],infile[80],outfile[80];
  float xnorm,ynorm;
  int xi,yi,z,xmin,ymin,xwide,ywide;

  printf("ReadFNG: reads .FNG file and normalizes positions\n");
  printf("base file name, d:\\fringe\\dat\\*.FNG assumed:");
  strcpy(basefilename,"d:\\fringe\\dat\\");
  scanf("%s",name);
  strcat(basefilename,name);
  strcpy(infile,basefilename);
  strcat(infile,".fng");
  strcpy(outfile,basefilename);
  strcat(outfile,".nfg");
  printf("reading file: %s\n",infile);
  printf("writing file: %s\n",outfile);

  fpin=fopen(infile,"r");
  if(fpin==NULL) {
    printf("error opening input fringe data file: %s\n",infile);
    exit(1);
  }
  fpout=fopen(outfile,"w");
  if(fpout==NULL) {
    printf("error opening output fringe data file: %s\n",outfile);
    exit(1);
  }

/* ------------------------------------------------------------------- */

  fscanf(fpin," %d %d %d %d",&xmin,&ymin,&xwide,&ywide);
  fscanf(fpin,"%d %d %d",&z,&xi,&yi);
  while (z>=0) {
    xnorm=(float)-1 +(float)2*(float)(xi-xmin)/(float)(xwide);
    ynorm=(float)1 -(float)2*(float)(yi-ymin)/(float)(ywide);
    fprintf(fpout,"%-6.3f\t%-6.3f\t%d\n",xnorm,ynorm,z);
    fscanf(fpin,"%d %d %d",&z,&xi,&yi);
  }
  fclose(fpin);
  fclose(fpout);
}
/* ********************************************************************** */
/* polyfit.c

fits a wavefront in the form of a power series in x,y to
a series of fringe positions

*/
/* ********************************************************************* */
#define MP 15
#define NP 15
#define SKIP 0

#include <stdio.h>
#include <process.h>
#include <malloc.h>
#include <math.h>
#include <string.h>
#include "\nr\nrutilh.h"
#include "\nr\nr.h"

/* ==================================================================== */
/* procedure declarations */

float poly(int m, float x, float y);
float polyfit(float *p, float x, float y);

/* ==================================================================== */
struct fringe
{
  float x;   /* x value of fringe point */
  float y;   /* y value of fringe point */
  float z;   /* order of fringe (integral but float for calculation) */
};

float poly(int m, float x, float y);

/* ********************************************************************* */
void main()
{
  FILE *fp;
  char basefilename[40],name[12],infile[80],outfile[80];

  int i,j,k,l,m,n,npts;
  float wmax,wmin,*w,*b,*x;
  float xval,yval,zval;
  float **a,**u,**v;
  float zero=(float)0;
  float sum,diff,rms;
  struct fringe *f;

  f=(struct fringe *)malloc((size_t)1000*sizeof(struct fringe));
  if(f==NULL) {
    printf("error allocating fringe data space\n");
    exit(1);
  }
  m=MP;
  n=NP;
  w=vector(1,NP);
  x=vector(1,NP);
  b=vector(1,MP);
  a=matrix(1,MP,1,NP);
  u=matrix(1,MP,1,NP);
  v=matrix(1,NP,1,NP);

  for(i=0;i<1000;i++) {
    f[i].x=zero;
    f[i].y=zero;
    f[i].z=zero;
  }
  for(j=1;j<=MP;j++) {
    w[j]=zero;
    x[j]=zero;
    b[j]=zero;
  }
  for(j=1;j<=NP;j++) {
    for(k=1;k<=NP;k++) {
      a[j][k]=zero;
      u[j][k]=zero;
      v[j][k]=zero;
    }
  }
```

```
/* --------------------------------------------------------------------- */
/* read fringe data
   This section reads an arbitrarily long list of data triplets:
   x-position  y-position  fringe order
*/

  printf("PolyFit: reads normalized fringe positions .NFG file\n");
  printf("         writes polynomial coefficient .POL file\n");
  printf("base file name, d:\\fringe\\dat\\*.NFG assumed:");
  strcpy(basefilename,"d:\\fringe\\dat\\");
  scanf("%s",name);
  strcat(basefilename,name);
  strcpy(infile,basefilename);
  strcat(infile,".nfg");
  strcpy(outfile,basefilename);
  strcat(outfile,".pol");
  printf("reading file: %s\n",infile);
  printf("writing file: %s\n",outfile);

  fp=fopen(infile,"r");
  if(fp==NULL) {
    printf("error opening fringe data file\n");
    exit(1);
  }
  i=0;
  while(!feof(fp)) {
    fscanf(fp," %f %f %f",&xval,&yval,&zval);
    f[i].x=xval;
    f[i].y=yval;
    f[i].z=zval;
    i++;
  }
  npts=i-1;
  if (ferror(fp)) {
    printf("error reading fringe data file\n");
    fclose(fp);
    exit(1);
  }
  fclose(fp);
  printf("%d fringe points read from file\n",npts);

/* --------------------------------------------------------------------- */
/* generate matrix and column vector for minimization.
   b is the column vector b_m = Sum_i z_i*poly(m,x_i,y_i).
   a is the matrix a_mn = Sum_i poly(m,x_i,y_i)*poly(n,x_i,y_i).
   Note that a_mn = a_nm.
   Solve [[a]][x]=[b] for the coefficents [x].
*/

  printf("generating matrix and vector for minimization\n");
  for(j=1;j<=15;j++) {
    sum=zero;
    for(i=0;i<npts;i++) sum+=f[i].z*poly(j,f[i].x,f[i].y);
    b[j]=sum;
    for(k=1;k<=15;k++) {
      sum=zero;
      for(i=0;i<npts;i++)
 sum+=poly(j,f[i].x,f[i].y)*poly(k,f[i].x,f[i].y);
      a[j][k]=sum;
    }
  }

/* --------------------------------------------------------------------- */
/*   Solve [[a]][x]=[b] for the coefficents [x]. */

  for(k=1;k<=MP;k++) for(l=1;l<=NP;l++) u[k][l]=a[k][l];

  printf("decomposing matrix\n");
  svdcmp(u,n,n,w,v);

/* find maximum singular value */

  wmax=(float)0.0;
  for (k=1;k<=n;k++) if (w[k] > wmax) wmax=w[k];
  wmin=wmax*(1.0e-6);
  for (k=1;k<=n;k++)
    if (w[k] < wmin){
      printf("setting value %d of %g to zero\n",k,w[k]);
      w[k]=(float)0.0;
    }
  printf("begin backsubstitution\n");
  svbksb(u,w,v,n,n,b,x);

  printf("solution vector\n\n");
  for(i=1;i<=NP;i++) printf("%d: %g\n",i,x[i]);
  printf("\n");

/* ------------------------------------------------------------------- */
/* calculate fit error */


  sum=zero;
  for(i=0;i<npts;i++) {
    diff=f[i].z-polyfit(x,f[i].x,f[i].y);
    sum+=diff*diff;
  }
  rms=(float)sqrt((double)sum)/(float)npts;
  printf("rms error= %g\n",rms);

/* ------------------------------------------------------------------- */
/* write polynomial coefficients to output file */

  fp=fopen(outfile,"w");
  if(fp==NULL) {
    printf("error opening polynomial data file\n");
    exit(1);
  }
  for(i=1;i<=NP;i++) fprintf(fp,"%g\n",x[i]);
  fclose(fp);

/* ------------------------------------------------------------------- */
/* clean up and go home */

  free(f);
  free_matrix(a,1,MP,1,NP);
  free_matrix(v,1,NP,1,NP);
  free_matrix(u,1,MP,1,NP);
  free_vector(b,1,NP);
  free_vector(x,1,NP);
  free_vector(w,1,NP);
}

/* ******************************************************************** */
float poly(int m, float x, float y)
{
  switch(m) {
  case 1: return (float)1;
   break;
  case 2: return x;
   break;
  case 3: return y;
   break;
  case 4: return x*x;
   break;
  case 5: return x*y;
   break;
  case 6: return y*y;
   break;
  case 7: return x*x*x;
   break;
  case 8: return x*x*y;
   break;
  case 9: return x*y*y;
   break;
  case 10: return y*y*y;
    break;
  case 11: return x*x*x*x;
    break;
  case 12: return x*x*x*y;
    break;
  case 13: return x*x*y*y;
    break;
  case 14: return x*y*y*y;
    break;
  case 15: return y*y*y*y;
    break;
  default: printf("f subscript out of bounds\n");
    break;
  }
}
/* ------------------------------------------------------------------- */
```

```
float polyfit(float *p, float x, float y)
{
  int m;
  float sum;

  sum=(float)0;
  for(m=1;m<=15;m++) {
    sum+=p[m]*poly(m,x,y);
  }
  return(sum);
}
/* ====================================================================== */
```